Proceedings of the

Second International Samos Workshop on Systems, Architectures, Modeling, and Simulation

SAMOS 2002

July 22-25, 2002
Research and Training
Institute of east Aegean
Samos, Greece

supported by

Dutch Technology Foundation (STW)

Copyright

SAMOS ISBN : 90 - 9016023 - X
SAMOS Order Number 2002-0722

Previous Publications

A selection of papers presented at the SAMOS 2001 workshop have been published in Springer's Lecture Notes in Computer Science LNCS 2268 *Embedded Processor Design Challenges*

Copy ordering

Additional copies of this proceedings may be ordered from:

SAMOS Initiative,
P.O. Box 9512
2300 Leiden, The Netherlands

Phone (+31 71) 527 5778
Fax (+31 71) 527 6985
E-mail edd@liacs.nl

Printed in the Netherlands

Table of contents

Processors

Abstract

Message from the Organizers

The SAMOS Initiative is an *International Informal Initiative* that emerged from de facto dependencies between leading research programs conducted in Maryland (University of Maryland), The Netherlands (Delft, Leiden, and Amsterdam Universities), Paderborn (Paderborn University), and Rennes (IRISA), headed by Shuvra Bhattacharyya, Stamatis Vassiliadis, Ed Deprettere, Andy Pimentel, Juergen Teich, and Patrice Quinton, respectively.

The SAMOS 2001 workshop emerged from an attempt to organize a visiting program among the above mentioned researchers: Why not meeting each other in one place instead of organizing point-to-point meetings? Stamatis Vassiliadis originating from the Island of Samos in Greece proposed to meet there, and so it happened.
The SAMOS 2001 informal workshop was a successful event, and the initiators decided to go for a 2002 workshop as well.
The workshop is by invitation only, and the selection criterion is that the chance for research co-operation is maximized.

Embedded systems design is an emerging field of research that addresses many challenging problems. No single expert is capable of covering the whole field and, therefore, co-operation is indispensable. Who can model applications; who can model architectures; who can model mappings of applications into architectures; who can perform exploration, and who can synthesize. Why would a researcher try to master all aspects of embedded systems design when an approach can be conceived in which abstraction layers can be assigned to layer experts. The Embedded systems design problem is a divide and rule game with a win-win flavor. The SAMOS Initiative aims at identifying the players and to get them involved in the game.

Modeling of Intra-task Parallelism in Sesame

Andy D. Pimentel, Frank P. Terpstra, Simon Polstra, and Joe E. Coffland

Computer Architecture & Parallel Systems group
Dept. of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{andy,ftrpstra,spolstra,jcofflan}@science.uva.nl

Abstract. The Sesame environment provides modeling and simulation methods and tools for the efficient design space exploration of heterogeneous embedded multimedia systems. It specifically targets the performance evaluation of embedded systems architectures in which task-level parallelism is available. In this paper, we present techniques that allow Sesame to also model intra-task parallelism exploited at the architecture level. Moreover, we describe a case study using a QR decomposition application to validate our modeling concepts. To this end, we were able to compare the performance estimates of our abstract system models with the results of an actual FPGA implementation. The results are promising in the sense that they show good accuracy with minimal modeling effort.

1 Introduction

Modern embedded systems, like those for media and signal processing, often have a heterogeneous system architecture, consisting of components in the range from fully programmable processor cores to dedicated hardware components. Increasingly, these components are integrated as a system-on-chip exploiting task-level parallelism in applications. Due to the high degree of programmability that is usually provided by such embedded systems, they typically allow for targeting a whole range of applications with varying demands. All of the above characteristics greatly complicate the design of these embedded systems, making it more and more important to have good tools available for exploring different design choices at an early stage in the design.

In the context of the Artemis (ARchitectures and meThods for Embedded MedIa Systems) project [19], we are developing an architecture workbench which provides modeling and simulation methods and tools for the efficient design space exploration of heterogeneous embedded multimedia systems. This architecture workbench should allow for rapid performance evaluation of different architecture designs, application to architecture mappings, and hardware/software partitionings and it should do so at multiple levels of abstraction *and* for a wide range of multimedia applications.

In this paper, our focus is on a prototype modeling and simulation environment, called Sesame [18]. According to the Artemis modeling methodology [19], this environment uses separate application models and architecture models and an explicit mapping step to map an application model onto an architecture model. This mapping is realized by means of trace-driven co-simulation of the application and architecture models, where the execution of the application model generates application events that represent the application workload imposed on the architecture. Application models consist of communicating parallel processes, thereby expressing the task-level parallelism available in the applications. By mapping the event traces generated by different application model processes onto the various system architecture components, this task-level parallelism is exploited at the architecture level. In addition, the underlying architecture may also exploit intra-task parallelism inside a single trace. This paper presents the newly added techniques Sesame applies to model architectures that exploit such intra-task parallelism. Moreover, using a case study with the QR decomposition algorithm as application, we demonstrate the effectiveness of our modeling methodology.

The remainder of this paper is organized as follows. Section 2 briefly describes related work in the area of modeling and simulation of complex embedded systems. Section 3 gives a general overview of the Sesame modeling and simulation environment, while in Section 4 we present a more detailed description of Sesame's synchronization layer. In Sections 5 and 6, we describe the methods applied to model intra-task parallelism and discuss their impact on Sesame's synchronization and architecture model layers. Section 7 presents some validation results we obtained from the case study with the QR decomposition application. Finally, Section 8 discusses several open issues and Section 9 concludes the paper.

2 Related work

Various research groups are active in the field of modeling and simulating heterogeneous embedded systems, of which some are academic efforts (e.g., [6, 11, 9]) and others commercial [8] and industrial efforts (e.g., [5]). Many efforts in this field *co-simulate* the software parts, which are mapped onto a programmable processor, and the hardware components and their interactions together in one simulation. Because an explicit distinction is made between software and hardware simulation, it must be known which application components will be performed in software and which ones in hardware before a system model is built. This significantly complicates the performance evaluation of different hardware/software partitioning schemes since a new system model may be required for the assessment of each partitioning.

A number of exploration environments, such as VCC [1], Polis [4] and eArchitect [2], facilitate more flexible system-level design space exploration by providing support for mapping a behavioral application specification to an architecture specification. Within the Artemis project, however, we try to push the separation of modeling application behavior and modeling architectural constraints at the system level to even greater extents. To this end, we apply trace-driven co-simulation of application and architecture models. Like was shown in [18], this leads to efficient exploration of different design alternatives while also yielding a high degree of reusability. The work of [15] also uses a trace-driven approach, but this is done to extract communication behavior for studying on-chip communication architectures. Rather than using the traces as input to an architecture simulator, their traces are analyzed statically. In addition, a traditional hardware/software co-simulation stage is required in order to generate the traces.

Finally, the Archer project [23] shows a lot of similarities with our work. This is due to the fact that both our work and Archer are spin-offs from the Spade project [17].

A major difference is, however, that Archer follows an entirely different application-to-architecture mapping approach. Instead of using event-traces, it maps symbolic programs, which are derived from the application model, onto architecture resources.

3 The Sesame modeling and simulation environment

The Sesame modeling and simulation environment [18], which builds upon the ground-laying work of the Spade framework [17], facilitates the performance analysis of embedded systems architectures in a way that directly reflects the so-called Y-chart design approach [13]. In Y-chart based design, a designer studies the target applications, makes some initial calculations, and proposes an architecture. The performance of this architecture is then quantitatively evaluated and compared against alternative architectures. For such performance analysis, each application is mapped onto the architecture under investigation and the performance of each application-architecture combination is evaluated. Subsequently, the resulting performance numbers may inspire the designer to improve the architecture, restructure the application(s) or modify the mapping of the application(s).

In accordance to the Y-chart approach, Sesame recognizes separate application and architecture models within a system simulation. An application model describes the functional behavior of an application, including both computation and communication behavior. The architecture model defines architecture resources and captures their performance constraints. Essential in this modeling methodology is that an application model is independent from architectural specifics, assumptions on hardware/software partitioning, and timing characteristics. As a result, a single application model can be used to exercise different hardware/software partitionings and can be mapped onto a range of architecture models, possibly representing different system architectures or simply modeling the same system architecture at various levels of abstraction. After mapping, an application model is co-simulated with an architecture model allowing for evaluation of the system performance of a particular application, mapping, and underlying architecture.

For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [12] in which parallel processes – implemented in a high level language – communicate with each other via unbounded FIFO channels. In the Kahn paradigm, reading from channels is done in a blocking manner, while writing is non-blocking. The computational behavior of an application is captured by instrumenting the code of each Kahn process with annotations which describe the application's computational actions. The reading from or writing to Kahn channels represents the communication behavior of a process within the application model. By executing the Kahn model, each process records its actions in order to generate a trace of application events, which is necessary for driving an architecture model. Initially, the application events typically are coarse grained, such as *execute(DCT)* or *read(pixel-block,channel_id)*, and they may be refined as the underlying architecture models are refined. We note that in the remainder of this paper, computational application events will be referred to as *execute events*.

To execute Kahn application models, and thereby generating the application events that represent the workload imposed on the architecture, Sesame features a process

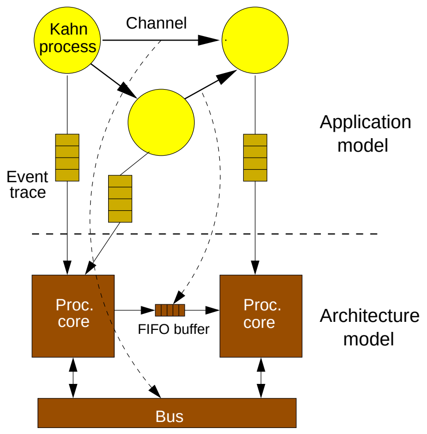


Fig. 1. Mapping a Kahn application model onto an architecture model.

network execution engine supporting Kahn semantics. This execution engine runs the Kahn processes as separate threads using the Pthreads package. For now, there is a limitation that the Kahn processes need to be written in C++. In the near future, C and Java support will be added. The structure of the application models (i.e., which processes are used in the model and how they are connected to each other) is described in a language called YML (Y-chart Modeling Language)[22]. This is an XML-based language which is similar to Ptolemy's MoML [16] but is slightly less generic in the sense that YML only needs to support a few simulation domains. As a consequence, YML only supports a subset of MoML's features. However, YML provides one additional feature in comparison to MoML as it contains built-in scripting support. This allows for loop-like constructs, mapping & connectivity functions, and so on, which facilitate the description of large and complex models.

The performance of an architecture can be evaluated by simulating the performance consequences of the incoming execute and communication events from an application model. This requires an explicit mapping of the processes and channels of a Kahn application model onto the components of the architecture model. The generated trace of application events from a specific Kahn process is therefore routed towards a specific component inside the architecture model by using a trace-event queue. This is illustrated in Figure 1. Since the application-model execution engine and the architecture simulator run as separate processes[1], these trace-event queues are currently implemented via Unix named-pipes. Alternative implementations of the queues, such as using shared memory, are foreseen in the future. If two or more Kahn processes are mapped onto a single architecture component (e.g., when several application tasks are mapped onto a

[1] Running the application-model execution engine as a separate process also makes it easy to analyze the application model in isolation. This can be beneficial as it allows for investigation of the upper bounds of the performance and may lead to early recognition of bottlenecks within the application itself.

microprocessor), then the events from the different trace-event queues need to be scheduled. The next section explains how this is done.

An architecture model solely accounts for architectural (performance) constraints and therefore does not need to model functional behavior. This is possible because the functional behavior is already captured in the application model, which subsequently drives the architecture simulation. An architecture model is constructed from generic building blocks provided by a library. This library contains template performance models for processing cores, communication media (like busses) and different types of memory. These template models can be freely extended and adapted. All architecture models in Sesame are implemented using a small but powerful discrete-event simulation language, called Pearl, which provides easy construction of the models and fast simulation [18]. The structure of architecture models – specifying which building blocks are used from the library and the way they are connected – is also described in YML.

4 The synchronization layer

When multiple Kahn application model processes are mapped onto a single architecture model component, the event traces need to be scheduled. For this purpose, Sesame provides an intermediate *synchronization layer*, which is illustrated in Figure 2. This layer guarantees deadlock-free scheduling of the application events and forms the application and architecture dependent structure that connects the architecture-independent application model with the application-independent architecture model. The synchronization layer, which can be automatically generated from the YML description of an application model, consists of virtual processor components and FIFO buffers for communication between the virtual processors. There is a one-to-one relationship between the Kahn processes in the application model and the virtual processors in the synchronization layer. This is also true for the Kahn channels and the FIFO channels in the synchronization layer, except for the fact that the buffers of the latter channels are limited in size. Their size is parameterized and dependent on the modeled architecture. A virtual processor reads in an application trace from a Kahn process and dispatches the events to a processing component in the architecture model. The mapping of a virtual processor onto a processing component in the architecture model is parameterized and thus freely adjustable. Currently, this virtual processor to architectural processor mapping is specified in the YML description of the architecture model. We are working, however, towards an approach in which this mapping is specified in a separate YML mapping description.

As can be seen from Figure 2, multiple virtual processors can be mapped onto a single processor in the architecture model. In this scheme, execute events are directly dispatched by a virtual processor to the processor model. The latter subsequently schedules the events originating from different virtual processors according to some given policy (FCFS by default) and models their timing consequences. For communication events, however, the appropriate buffer at the synchronization layer is first consulted to check whether or not a communication is safe to take place so that no deadlock can occur. Only if it is found to be safe (i.e., for read events the data should be available and for write events there should be room in the target buffer), then communication events

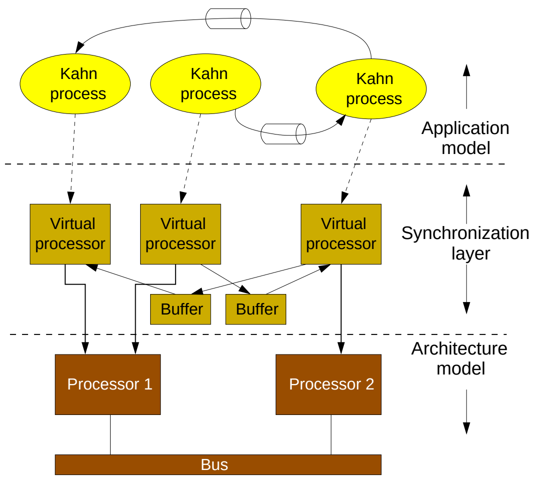


Fig. 2. The three layers within Sesame: the application model layer, the architecture model layer, and the synchronization layer which interfaces between application and architecture models.

may be dispatched to the processor component in the architecture model. As long as a communication event cannot be dispatched, the virtual processor blocks. This is possible because the synchronization layer is, like the architecture model, implemented in the Pearl simulation language and executes in the same simulation-time domain as the architecture model. As a consequence, the synchronization layer accounts for synchronization delays of communicating application processes mapped onto the underlying architecture, while the architecture model accounts for the computational latencies and the pure communication latencies (i.e., how long does it take to transfer an amount of data from X to Y). Each time a virtual processor dispatches an application event (either computation or communication) to a processor in the architecture model, it is blocked (in simulated time) until the event's simulation at the architecture level has finished.

The idea of concentrating synchronization behavior in a synchronization layer and separating it from (the latencies caused by) data transmission behavior is somewhat similar to the synchronization graph concept of [20]. However, our synchronization layer seems to be more flexible since it is dynamically scheduled and behaves like a ”Kahn” process network in which the FIFO buffers are bounded. As a consequence of the dynamic scheduling of the synchronization layer and the architecture model (remember that they both are executed in the same discrete-event simulation domain), dynamics at the architecture level such as contention can easily be taken into account within the synchronization layer.

5 Exploiting intra-task parallelism

Initially, Sesame only modeled the architecture's processing cores as black boxes which sequentially simulate the timing consequences of the incoming (linear) trace of application events. However, the architecture under investigation may also want to exploit

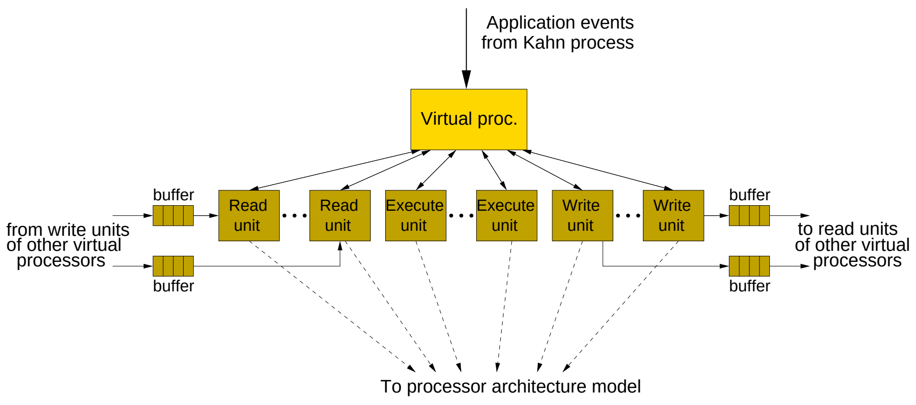


Fig. 3. Refining a virtual processor in the synchronization layer.

intra-task parallelism which is present in a single event trace from a Kahn application process. For example, a processing element may have multiple communication units which allow for performing independent reads and writes in parallel, or it may have multiple execution units for concurrently processing independent computations. To support the modeling and simulation of such intra-task parallelism, we extended Sesame's model library with component models that allow for refining the virtual processors in the synchronization layer and the processor components within the architecture models.

Figure 3 shows how a virtual processor in the synchronization layer, like the ones depicted in Figure 2, is refined. The virtual processor component now acts as a front-end to a range of (virtual) functional units. These functional units consist of read, write and execution units which can operate in parallel. The new virtual processor component has a symbolic-instruction window of parameterizable size in which it stores incoming application events and with which it analyzes them for parallel execution. According to the event type (execute event type, channel from/to which is read/written, etc.), the virtual processor dispatches incoming events to the appropriate functional unit. The number of entries in the symbolic-instruction window limits the number of outstanding (dispatched but not finished) events in the virtual processor. A window size of one implies sequential handling of the application events. In Figure 3, the arrows from the functional units back to the virtual processor refer to the acknowledgments the functional units transmit whenever the simulation of an event has finished.

The read and write units are connected via buffers[2] with other virtual processors, like discussed in Section 4, in order to establish the modeling of synchronizations between Kahn application processes in accordance to their mapping onto the underlying architecture. Hence, the read and write units do not dispatch a communication event to the architecture model unless it is safe to do so, i.e., the event cannot cause a deadlock. In addition, the execution and write units do not dispatch their incoming application events to the architecture model before all dependencies for these events are resolved.

[2] Per read or write unit, there may be multiple buffers connected.

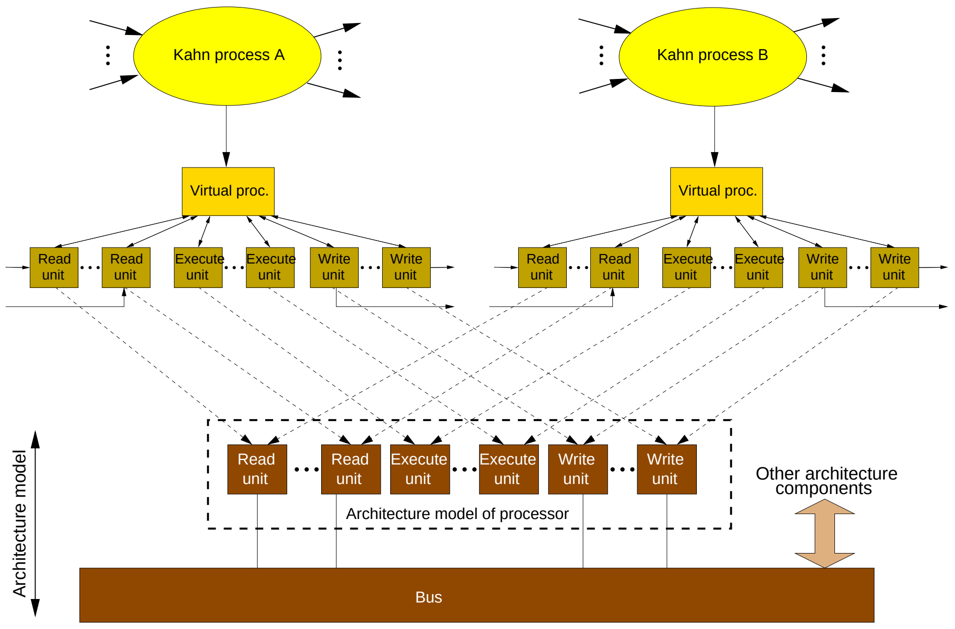


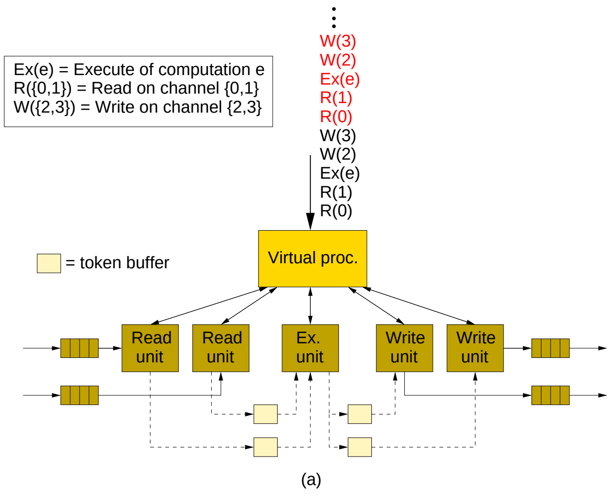
Fig. 4. Mapping multiple refined virtual processors onto a refined processor architecture model.

We will elaborate on this issue in the next section, which discusses the internal synchronizations within a refined virtual processor component.

Figure 4 illustrates how the refined virtual processors can be mapped onto a processor component in the architecture model which has been refined as well. The read units from the virtual processors that are mapped onto the same processor at the architecture level, are connected to the read units of the processor in the architecture model. Likewise, the virtual execution units are connected to the execution units of the processor architecture model, and so on. The functional units in the architecture model may again be black-box models which sequentially account for the timing consequences of the incoming application events dispatched by the synchronization layer. Alternatively, they may also be further refined. For example, a refined execution unit may model internally pipelined execution of execute events. Furthermore, in the example of Figure 4 all communication units in the architecture model are connected to a bus model. In reality, communication units within the architecture model may have different connections with each other (directly across a bus or via shared memory, point-to-point, etc.).

6 Dataflow for functional unit synchronization

To properly model parallel execution of application events from a single event trace, the dependencies between the events should be taken into account. For example, an execution unit in the synchronization layer may only dispatch an execute event to the execution unit in the architecture model when the read events it depends on have been sim-



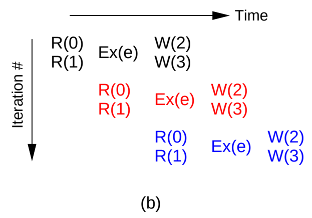


Fig. 5. Dataflow-based synchronization to resolve dependencies between functional units in a virtual processor. The architecture shown in (a) exploits pipeline parallelism, which is illustrated in (b).

ulated and delivered the required input for the execute. Likewise, a write event may be dispatched to the architecture model when it is safe to do so and when the read/execute events it depends on have been simulated.

Consider the example in Figure 5(a) in which a virtual processor is shown for a processor architecture with a pipeline of two read units, one execution unit and two write units. In this example, the trace generating Kahn process reads/writes from/to two channels which are mapped onto separate read and write units. The execute events in this example are dependent on the two preceding read events, while the two write events are dependent on the preceding execute event. In Figure 5(b) the resulting pipeline parallelism is illustrated.

The synchronization between the functional units in order to resolve dependencies is done via buffered token channels. In Figure 5(a), for example, the read units have a token channel to the execution unit. A read unit sends a token along its token channel whenever a read event finished, i.e., has been simulated at architecture level. The size of the token channel's buffer determines how far the read unit can run ahead, or in other words, the amount of internal buffering a read unit has. If the token channel's buffer is

```
class v_read_unit

[...]

sig_room : (unit_id : integer)
{ }

read : ()
{
   block(sig_room);              // block until there’s room in token buffer
   input_buffer ! get();         // wait until there’s data in input FIFO
   ex_unit !! sig_data(unit_id); // send token to execution unit
   virt_proc !! op_done();       // signal completion to virt. processor
}

{
   while (1) {
      block( read, signal_room );
   }
}
```

Fig. 6. Pearl code for a read unit object from the synchronization layer.

full, then the read unit stalls until the execution unit has removed one or more tokens from the channel’s buffer. During such a stall, a read unit cannot handle new read events.

In our example, the execution unit reads the tokens generated by the read units. Associated with each execute event type, there are two *bitmaps*. The first one describes on which token channels the particular execute event is dependent, i.e., which read units produce data needed by the execute event. The second bitmap describes which functional units are dependent on the execute event. So, it relates to output token channels.

The execution unit must have received a token from all of the required token channels, implying that dependencies have been resolved, before the execute event may be dispatched to the architecture model. Likewise, after an execute event has been simulated at the architecture level, the execution unit sends tokens along the required output token channels (as specified by the second bitmap). As a consequence, the write units, which are waiting for tokens from the execution unit, are enabled to dispatch dependent write events to the architecture model. To summarize, synchronizations due to dependencies between functional units in the synchronization layer are handled using the dataflow principle with token transmissions between the functional units. To be more specific, this dataflow mechanism adheres to integer-controlled dataflow [7]. Of course, the placement of token channels between functional units and their buffer sizes are freely adjustable. For the time being, however, we slightly restricted the choice of functional units as we currently assume that there can be only one execution unit per processor. In Section 8, we come back to this issue and indicate how our modeling concepts may be extended to support multiple execution units per processor.

To give an impression of how the implemented models look like, Figure 6 shows the Pearl code for a read unit from the synchronization layer (the variable declarations have been omitted). As Pearl is an object-based language and architecture components are modeled by objects, the code shown in Figure 6 embodies the class of read unit objects. As the explanation of the code is beyond the scope of this paper, the interested reader is referred to [18] for a more detailed discussion of a Pearl code sample.

In our implementation, it is straightforward to change the policy defining when token buffers can be read from or written to. More specifically, a functional unit can wait until all of its required tokens are available before it retrieves the tokens from the buffers or it can retrieve a required token whenever it becomes available. In the latter case, the producer of the token may be unblocked earlier and thereby allowing it to proceed with processing new application events.

We note that the synchronizations between functional units are only performed in the synchronization layer and are not needed within the underlying architecture model. This is because once application events are dispatched from the synchronization layer to the architecture model, they are safe to simulate, i.e., they cannot cause deadlocks and their dependencies have been resolved. This scheme nicely fits our approach in which all synchronization overheads are accounted for in the synchronization layer.

7 A case study: QR decomposition

To validate the previously presented concepts on how to model the exploitation of intra-task parallelism, we have performed a case study using a set of application model instances of the well-understood QR decomposition algorithm. These application models are the result of the Compaan work [14] done at Leiden University. The Compaan tool is able to automatically generate Kahn application models from nested loop programs written in Matlab, which in our case is the QR decomposition algorithm. In addition, it can perform code transformations such as loop unrolling to increase task-level parallelism inside applications [21].

The Kahn application models generated by the Compaan tool are suitable for a direct implementation in hardware on an FPGA. For this purpose, application models are translated into VHDL [10]. This gives us the unique opportunity to validate our abstract architecture models against an actual FPGA implementation. In the VHDL implementation of a Kahn application model, pre-defined node components are connected in a network. This is done according to the connections between the processes in the application model. The node components, which represent the functional behavior of the Kahn processes in the application model, are implemented in a pipelined fashion that is similar to the one shown in Figure 5. Conceptually, this means that each node component contains a number of read and write units and a single execution unit. So, besides exploiting task-level parallelism by the VHDL network of node components, each node component also exploits intra-task parallelism using its internally pipelined architecture.

Regarding the QR application, we studied five different instances of its application model generated by Compaan. In each instance, the loops in the code have been unrolled a different number of times. This loop unrolling creates new Kahn processes, thereby increasing the task-level parallelism available in the application [21]. Additional information on the Kahn application model of the QR decomposition algorithm can be found in [10]. For each of the application model instances, we described the structure of the application model in YML to be able to run the model with Sesame's application-model execution engine. As a side-note, it is worth mentioning that the generation of the YML descriptions of the application model instances is performed fully automatically by means of a visitor tool.

Our Sesame architecture model, onto which the QR application model instances are mapped, is similar to the VHDL implementation of a Kahn application model in the sense that it also consists of processor components connected in a network with a topology identical to that of the application model. Each processor component is modeled with our refined (virtual) processor model (see Section 5) and uses the pipelined architecture as shown in Figure 5(a). Between processor components in the architecture model there are point-to-point FIFO channels.

Recall that the structure of Sesame's architecture models is described in YML. Because of YML's built-in scripting support, this allowed us to construct a generic reusable template for the refined (virtual) processor model. The processor network in the architecture model is thus obtained by repetitively instantiating this template with possibly different parameters and linking these processor instances together according to the topology of the application model. This topology information is derived from our YML description of the Kahn application model.

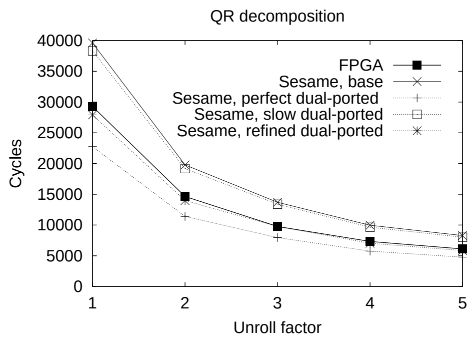
7.1 Experiments

Our first experiments were performed using a Sesame synchronization layer and architecture model with the following characteristics. The size of the FIFO buffers is 256 elements, which guarantees deadlock-free execution of the studied application model instances [10]. The functional units of processor components as well as the FIFO buffers are modeled as black boxes. Read and write operations to the FIFO buffers take 3 cycles each as specified in [10], while all execute events[3] are handled in a single cycle. The latter reflects the performance of a fully-utilized internal execution pipeline with a single-cycle throughput. Moreover, the token channels between the functional units at the synchronization layer have single-entry buffers. This means that the read and execution units cannot produce more than one result before consumption, i.e., they have only limited internal buffering.

In Figure 7(a), the performance of the FPGA implementation (modeled in VHDL) of the five QR application instances – with loop unroll factors of one to five – is shown. The figure also shows the performance estimates of our black-box Sesame model for these application model instances. These results are referred to as the *base* model in Figure 7. As shown in Figure 7(b), the black-box model yields an average error of 36% and a worst-case error of 40% with respect to the performance results of the FPGA implementation. The Sesame (base) performance estimates show the correct trend behavior but are consistently more pessimistic than those for the FPGA.

According to [10], the FPGA buffer implementation is based around a dual-ported RAM, where our base model models single-ported buffers. This explains why the results of the base model are pessimistic. As a next step, we ”opened up” the black-box FIFO model and adapted it to include dual-ported behavior. To this end, we modeled three variants of dual-ported FIFO buffers. Two of these variants represent implementation extremes, while the third one reflects the performance behavior of the actual FPGA

[3] In the QR application model, the execute events consist of vectorize and rotate operations.



(a)

Difference %	Base model	Dual-ported model		
		Perfect FIFO	Slow FIFO	Refined FIFO
Average	36	-21	32	-3.5
Worst case	40	-22	37	-4.7

(b)

Fig. 7. Validation results of our Sesame models for the QR decomposition application against the results from an actual FPGA implementation. The graph in (a) shows the (estimated) performance for five application instances with different loop unroll factors. The table in (b) gives the differences (in %) between estimates from our models and the FPGA numbers.

implementation. The results of these three dual-ported FIFO models are also shown in Figure 7. The curve labeled with *perfect dual-ported* shows the performance estimates when modeling the FIFO buffers as being perfectly dual-ported. The latter means that read and write operations on a buffer can be performed entirely in parallel, even when the buffer is empty. So, when receiving a read request in the empty buffer state, the read is blocked until a write request is coming in after which the incoming (written) data is immediately forwarded to the reading party. Consequently, both read and write latencies are entirely overlapped.

At the other extreme, the curve labeled with *slow dual-ported* in Figure 7 shows the Sesame performance estimates when modeling dual-ported FIFO buffers which are entirely sequential at the empty state. So, when receiving a read request in the empty buffer state, the read is blocked until a write has occurred and finished writing its data into the buffer (in our model, this takes 3 cycles).

Finally, the curve labeled with *refined dual-ported*, shows the Sesame results when incorporating more detailed knowledge on the actual FPGA buffer implementation into our model. Details on the FPGA implementation indicated that a monolithic 3-cycle read/write latency for the FIFO buffers does not reflect the actual behavior. In reality, the throughput at both sides of a FIFO buffer is 1 operation per 3 cycles, while the read latency turned out to be only 1 cycle. In our *refined dual-ported* model we have therefore split the 3-cycle delay into three 1-cycle delays and placed them at the appropriate places according to specification of the FPGA buffer implementation. This means that we refined the timing within our model while keeping its abstract structure intact.

Three important conclusions can be drawn from the results in Figure 7. First, the results reconfirm the modeling flexibility of Sesame. This is because we were able to model the three dual-ported buffer designs by changing less than ten lines in the code of the base model. Second, the results from the ‘perfect’ and ‘slow’ models – representing the two FIFO buffer implementation extremes – immediately indicate that the average accuracy of Sesame's performance estimates must lie in the range of -21% and +32%. In fact, our 'refined' model demonstrates how close our performance estimates can approximate reality since it yields an average error of only 3.5% and a worst case error of 4.7%. Knowing that Sesame targets performance evaluation in an early design stage and therefore models at a high level of abstraction, these accuracy numbers are very promising. Third, our results indicate that the studied hardware implementations of the QR decomposition application are highly sensitive to different FIFO buffer designs. Since the performance estimates of the 'perfect' buffer model show a speedup of 68% over the results of the 'slow' buffer model, the handling of the empty state in the FIFO buffer seems to be an important design issue.

Since Sesame targets performance evaluation in an early design stage, where the design space that needs to be explored typically is very large, the required modeling effort and the simulation speed of Sesame is worth noting. The architecture models in this case study, including the components in the synchronization layer, consist of less than 500 lines of Pearl code. It takes Sesame about 16 seconds on a 333MHz Sun Ultra 10 to perform the architecture simulation for all five application model instances in a batch.

8 Discussion

So far, we have assumed that in the set of functional units of a refined (virtual) processor there is only one execution unit. Processing cores, however, might have multiple execution units that can perform computations in parallel. We are currently investigating whether or not our dataflow approach is sufficient for dealing with dependencies between execution units. In any case, for such inter-execution dependencies we need to extend our dataflow scheme such that tokens are typed, like in the tagged-token model [3]. With the typed tokens, an execution unit can differentiate between the production of results from different execute event types. To support such typed tokens, the bitmaps need to be extended from single-bit values to multiple-bit values to be able to specify which token types are required for an application event.

Moreover, we currently use static bitmaps per execute event type. We found, however, that this causes problems when, for example, execute events of the same type require data from different read units in different stages of the application model's execution. This can be solved by dynamically adding the bitmap information to the execute events in the traces.

9 Conclusions

In this paper, we presented the techniques applied by the Sesame modeling and simulation environment to model intra-task parallelism exploited at the architecture level for task-parallel applications. To this end, our processor models are refined to the level of functional units which can operate in parallel and which are synchronized to resolve dependencies by means of a dataflow mechanism. Using a case study, in which we were able to compare our simulation results with the results from an actual FPGA implementation, we demonstrated that our modeling methodology is flexible and shows good accuracy.

Acknowledgments

This research is supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW. We thank the people from the Embedded Real-time Computing group at Leiden University, and in particular Todor Stefanov, Bart Kienhuis, Vladimir Živković, Laurentiu Nicolae and Ed Deprettere, for providing the application models for the QR case study and for their invaluable feedback on this work. In addition, we would like to thank Tim Hariss from QinetiQ Ltd (UK) for providing us with the details on the FPGA implementation of the QR application.

References

1. Cadence Design Systems, Inc., http://www.cadence.com/.
2. Innoveda Inc., http://www.innoveda.com/.
3. Arvind and K. P. Gostelow. The U-Interpreter. *IEEE Computer*, 15(2):42–49, Feb. 1982.
4. F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS approach*. Kluwer Academic Publishers, 1997.
5. J.-Y. Brunel, E.A. de Kock, W.M. Kruijtzer, H.J.H.N. Kenter, and W.J.M. Smits. Communication refinement in video systems on chip. In *Proc. 7th Int. Workshop on Hardware/Software Codesign*, pages 142–146, May 1999.
6. J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, Apr. 1994.
7. J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer valued control streams. In *Proc. of the 28th Asilomar conference on Signals, Systems, and Computers*, Oct. 1994.
8. P. Dreike and J. McCoy. Co-simulating software and hardware in embedded systems. *Embedded Systems Programming*, 10(6), June 1997.
9. R.K. Gupta, C.N. Coelho Jr., and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proc. of the Design Automation Conference*, pages 225–230, June 1992.
10. T. Harriss, R. Walke, B. Kienhuis, and E.F. Deprettere. Compilation from Matlab to process networks realized in FPGA. In *Proc. of the 35th Asilomar conference on Signals, Systems, and Computers*, Nov. 2001.
11. K. Hines and G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proc. of the Design Automation Conference*, pages 395–400, June 1997.
12. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, 1974.
13. B. Kienhuis, E.F. Deprettere, K.A. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proc. of the Int. Conf. on Application-specific Systems, Architectures and Processors*, July 1997.

14. B. Kienhuis, E. Rijpkema, and E.F. Deprettere. Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In *Proc. of the 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, May 2000.
15. K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(6):768–783, June 2001.
16. E. A. Lee and S. Neuendorffer. MoML - a Modeling Markup Language in XML, version 0.4. Technical Report UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley, March 2000.
17. P. Lieverse, P. van der Wolf, E.F. Deprettere, and K.A. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3):197–207, November 2001. Special issue on SiPS'99.
18. A. D. Pimentel, S. Polstra, F. Terpstra, A. W. van Halderen, J. E. Coffland, and L.O. Hertzberger. Towards efficient design space exploration of heterogeneous embedded media systems. In *Proc. of the 1st Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS)*, pages 57–73, July 2001.
19. A.D. Pimentel, P. Lieverse, P. van der Wolf, L.O. Hertzberger, and E.F. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, Nov. 2001.
20. S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.
21. T. Stefanov, B. Kienhuis, and E.F. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Proc. of the 10th Int. Symposium on Hardware/Software Codesign (CODES'02)*, pages 7–12, May 2002.
22. F. Terpstra. Design considerations and description of YML. Technical report, Univerity of Amsterdam, Jan. 2002.
23. V. Živković, P. van der Wolf, E.F. Deprettere, and E.A. de Kock. Design space exploration of streaming multiprocessor architectures. To appear in the *Proc. of the IEEE Workshop on Signal Processing Systems (SIPS'02)*, Oct. 2002.

In *Proceedings of the Second International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS 2002)*, Island of Samos, Greece, July 22–25, 2002.

Energy Estimation for Piecewise Regular Processor Arrays*

Frank Hannig and Jürgen Teich

University of Paderborn, D-33098 Paderborn, Germany,
{hannig, teich}@date.upb.de,
URL: http://www-date.upb.de

Abstract. In this paper, we present a first approach for array-level energy estimation during high-level synthesis when mapping piecewise regular algorithms onto massively parallel full size processor arrays. Innately, piecewise regular algorithms have some power consumption friendly properties, e.g., they may be mapped onto processor arrays with only local interconnect and memory. In addition to these properties, we show that the chosen mapping has a significant influence on the power consumption. Our energy estimation approach identifies regions with decreased switching activity of functional units' input operands. For these regions with reduced activity, a lower power consumption can be directly obtained from a generated table based model. Experimental results fortify the accuracy and efficiency of our methodology.

1 Introduction

Nowadays, low power has become an important design criterion last but not least due to all the mobile phones and portable computers. These devices have to handle increasingly computational-intensive algorithms like video processing (MPEG4) or other digital signal processing tasks (3G), but on the other hand they are limited in their power budget. The next generation of ULSI chips will allow to implement arrays of hundreds 32-bit micro-processors and more on a single die. Hence, parallelization techniques and compilers will be of utmost importance in order to map computational-intensive algorithms efficiently to these processor arrays.

In this context, our paper deals with the specific problem of estimating the power consumption when mapping a certain class of loop-specified computations called *piecewise regular algorithms* [24] onto a dedicated processor array. This work can be classified to the area of loop parallelization in the polytope model [8,15].

The rest of the paper is structured as follows. In Section 2, a brief survey of previous work on low power is presented. Section 3 introduces the class of

* Supported in part by the German Science Foundation (DFG) Project SFB 376 "Massively Parallel Computation".

algorithms we are dealing with. In Section 4, we examine the power consumption of functional units in dependence on their input activity. Afterwards, an energy estimation methodology when mapping regular algorithms to processor arrays is described. The methodology and some results are discussed in Section 5. Future extensions and concluding remarks are presented in Section 6.

2 Related Work

A lot of previous work in the area of low power design during high-level synthesis has dealt with the issue of power estimation. Various methodologies for generating accurate models for datapath power consumption were presented.

In general these power estimation techniques can be divided into simulative and non-simulative categories. The non-simulative method in [16] estimates the power consumption from an information theoretical point of view. In [14], the authors describe a strategy called Dual Bit Type (DBT) model where not only the random activity of the least significant bits, but also the correlated activity of the most significant bits is taken into account. The method in [10] proposes a modeling approach for functional units that are typically used in digital signal processing systems, such as adders, multipliers and delay elements. Thereby, a 4-dimensional table-based [9] macro model is used by the authors.

Also, some works [3] focused on transformations at the algorithmic and the architectural level to obtain low power designs. In [2], transformations for nested loop programs are discussed. In [4, 18, 19, 21], several scheduling and binding techniques for low power are studied. Some energy estimations for processor arrays with hierarchical memory structures are made in [6].

However, to the best of our knowledge, our work presented here is the first which considers the relationship between space-time mappings of computation intensive algorithms and energy consumption. Here, we specify a power-consumption model used in the methodology described afterwards for energy estimation of piecewise regular processor arrays.

3 Notation and Background

3.1 Algorithms

The class of algorithms we are dealing with in this paper is a class of recurrence equations defined as follows:

Definition 1. *(Piecewise Regular Algorithm). A piecewise regular algorithm contains N quantified equations*

$$S_1[I], \ldots, S_i[I], \ldots, S_N[I]$$

Each equation $S_i[I]$ is of the form

$$x_i[I] \;=\; f_i(\ldots, x_j[I - d_{ji}], \ldots)$$

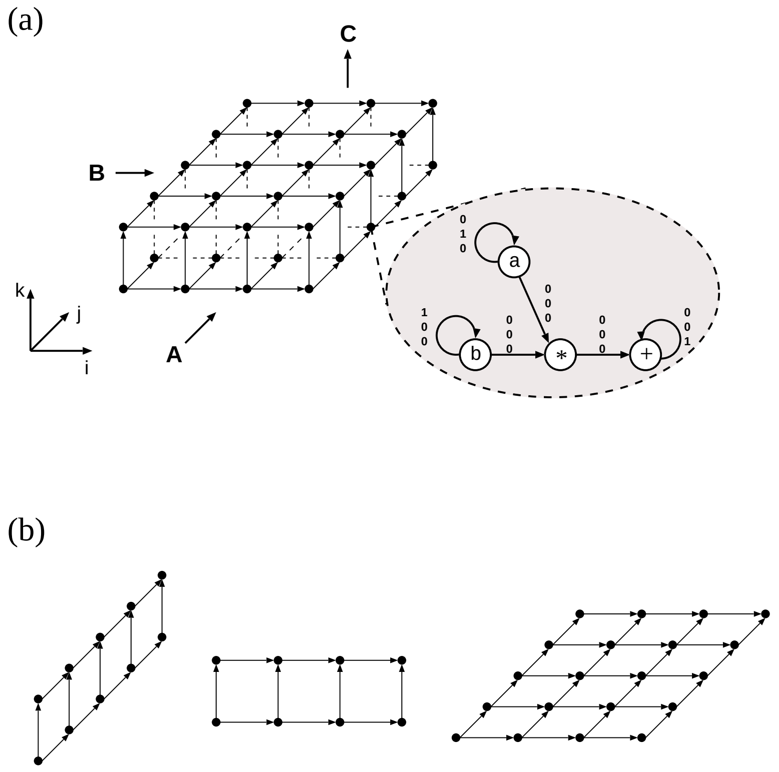


Fig. 1. In (a), an index space and the reduced dependence graph is shown. Some possible mappings are depicted in (b).

where $I \in \mathcal{I}_i \subseteq \mathbb{Z}^n$, $x_i[I]$ are indexed variables, f_i are arbitrary functions, $d_{ji} \in \mathbb{Z}^n$ are constant data dependence vectors, and ... denote similar arguments.

The domains $\mathcal{I}_i$ are called index spaces, and in our case defined as follows:

Definition 2. *(Linearly Bounded Lattice). A linearly bounded lattice denotes an index space of the form*

$$\mathcal{I} \;=\; \{I \in \mathbb{Z}^n \mid I = M\kappa + c \;\wedge\; A\kappa \geq b\}$$

where $\kappa \in \mathbb{Z}^l$, $M \in \mathbb{Z}^{n\times l}$, $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m\times l}$ and $b \in \mathbb{Z}^m$. $\{\kappa \in \mathbb{Z}^l \mid A\kappa \geq b\}$ defines an integral convex polyhedron or in case of boundedness a polytope in $\mathbb{Z}^l$. This set is affinely mapped onto iteration vectors I using an affine transformation ($I = M\kappa + c$).

Throughout the paper, we assume that the matrix M is square and invertible. Then, each vector κ is uniquely mapped to an index point I. Furthermore, we require that the index space is bounded.

For illustration purposes throughout the paper, the following example is used.

Example 1. The well known matrix multiplication algorithm computes the product $C = A \cdot B$ of two matrices $A \in \mathbb{R}^{N_1 \times N_3}$ and $B \in \mathbb{R}^{N_3 \times N_2}$ and is defined as follows

$$c_{ij} = \sum_{k=1}^{N_3} a_{ik} b_{kj} \quad \forall\ 1 \leq i \leq N_1 \ \wedge\ 1 \leq j \leq N_2.$$

A corresponding piecewise regular algorithm is given by

$$\begin{array}{ll}
\textit{input operations} & \\
\quad a[i,0,k] \leftarrow a_{ik} & 1 \leq i \leq N_1 \ \wedge\ 1 \leq k \leq N_3 \\
\quad b[0,j,k] \leftarrow b_{kj} & 1 \leq j \leq N_2 \ \wedge\ 1 \leq k \leq N_3 \\
\quad c[i,j,0] \leftarrow 0 & 1 \leq i \leq N_1 \ \wedge\ 1 \leq j \leq N_2 \\
\textit{computations} & \\
\quad a[i,j,k] \leftarrow a[i,j-1,k] & \forall (i\ j\ k)^{\mathrm{T}} = I \in \mathcal{I} \\
\quad b[i,j,k] \leftarrow b[i-1,j,k] & \forall (i\ j\ k)^{\mathrm{T}} = I \in \mathcal{I} \\
\quad z[i,j,k] \leftarrow a[i,j,k] \cdot b[i,j,k] & \forall (i\ j\ k)^{\mathrm{T}} = I \in \mathcal{I} \\
\quad c[i,j,k] \leftarrow c[i,j,k-1] + z[i,j,k] & \forall (i\ j\ k)^{\mathrm{T}} = I \in \mathcal{I} \\
\textit{output operations} & \\
\quad c_{ij} \qquad \leftarrow c[i,j,N_3] & 1 \leq i \leq N_1 \ \wedge\ 1 \leq j \leq N_2
\end{array}$$

The data dependence vectors are $d_{aa} = (0\ 1\ 0)^{\mathrm{T}}$, $d_{bb} = (1\ 0\ 0)^{\mathrm{T}}$, $d_{cc} = (0\ 0\ 1)^{\mathrm{T}}$, $d_{az} = (0\ 0\ 0)^{\mathrm{T}}$, $d_{bz} = (0\ 0\ 0)^{\mathrm{T}}$, and $d_{zc} = (0\ 0\ 0)^{\mathrm{T}}$. The index space is given by

$$\mathcal{I} = \{I = (i\ j\ k)^{\mathrm{T}} \in \mathbb{Z}^3 \mid 1 \leq i \leq N_1 \ \wedge\ 1 \leq j \leq N_2 \ \wedge\ 1 \leq k \leq N_3\}.$$

Computations of piecewise regular algorithms may be represented by a *dependence graph* (DG). The dependence graph of the algorithm of Example 1 is shown in Fig. 1 (a). The dependence graph expresses the partial order between the operations. Each variable of the algorithm is represented at every index point $I \in \mathcal{I}$ by one node. The edges correspond to the data dependencies of the algorithm. They are *regular* throughout the algorithm, i.e., $a[i,j,k]$ is directly dependent on $a[i,j-1,k]$. The dependence graph specifies implicitly all legal execution orderings of operations: if there is a directed path in the dependence graph from one node $a[J]$ to a node $z[K]$ where $J, K \in \mathcal{I}$, then the computation of $a[J]$ must precede the computation of $z[K]$.

Henceforth, and without loss of generality[1], we assume that all indexed variables are embedded in a common index space $\mathcal{I}$. Then, the corresponding dependence graphs can be represented in a reduced form.

Definition 3. *(Reduced Dependence Graph). A* reduced dependence graph *(RDG) $G = (V, E, D)$ of dimension n is a network where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. To each edge $e = (v_i, v_j)$ there is associated a dependence vector $d_{ij} \in D \subset \mathbb{Z}^n$.*

The RDG of the matrix multiplication algorithm is shown in Fig. 1 (a). Each node v in the graph corresponds to one equation in the section computations of the algorithm.

[1] All described methods can also be applied for each quantification individually.

3.2 Space-Time Mapping

Linear transformations as in Eq. (1), are used as *space-time mappings* [12,17] in order to assign a *processor index* $p \in \mathbb{Z}^{n-1}$ (space) and a *sequencing index* $t \in \mathbb{Z}$ (time) to index vectors $I \in \mathcal{I}$.

$$\begin{pmatrix} p \\ t \end{pmatrix} = TI = \begin{pmatrix} Q \\ \lambda \end{pmatrix} I \tag{1}$$

In Eq. (1), $Q \in \mathbb{Z}^{(n-1)\times n}$ and $\lambda \in \mathbb{Z}^{1\times n}$. The main reasons for using linear allocation and scheduling functions is that the data flow between PEs is local and regular which is essential for low power VLSI implementations. The interpretation of such a linear transformation is as follows: The set of operations defined at index points $\lambda \cdot I =$ const. are scheduled at the same time step. The index space of allocated processing elements (*processor space*) is denoted by $\mathcal{Q}$ and is given by the set $\mathcal{Q} = \{p \mid p = Q \cdot I \ \wedge \ I \in \mathcal{I}\}$. This set can also be obtained by choosing a projection of the dependence graph along a vector $u \in \mathbb{Z}^n$, i.e., any coprime[2] vector u satisfying $Q \cdot u = 0$ [12] describes the allocation equivalently.

Allocation and scheduling must satisfy that no data dependencies in the DG are violated. This is ensured by the following *causality constraint*

$$\lambda \cdot d_{ij} \geq 0 \quad \forall (v_i, v_j) \in E. \tag{2}$$

A sufficient condition for guaranteeing that no two or more index points are assigned to a processing element at the same time step is given by

$$\text{rank} \begin{pmatrix} Q \\ \lambda \end{pmatrix} = n. \tag{3}$$

Using the projection vector u satisfying $Q \cdot u = 0$, this condition is equivalent to $\lambda \cdot u \neq 0$ [24].

Definition 4. *(Iteration Interval) [26]. The* iteration interval π *of an allocated and scheduled piecewise regular algorithm is the number of time instances between the evaluation of two successive instances of a variable within one processing element.*

Definition 5. *(Block Pipelining Period) [13]. The* block pipelining period *of an allocated and scheduled piecewise regular algorithm is the time interval between the initiations of two successive problem instances and is denoted by β.*

Lets consider the matrix multiplication algorithm introduced in Example 1 as a problem instance. The whole matrices A and B have to read into the processor array before the next pair can be read, the time between these input operations is the block pipelining period β. Let λ be the schedule vector. Then, the block pipelining period β may be computed as follows,

$$\beta = \max_{I_1 \in \mathcal{I}} \{\lambda \cdot I_1\} \ - \ \min_{I_2 \in \mathcal{I}} \{\lambda \cdot I_2\} \ = \max_{I_1, I_2 \in \mathcal{I}} \{\lambda (I_1 - I_2)\}.$$

[2] A vector x is said to be *coprime* if the absolute value of the greatest value of the greatest common divisor of its elements is one.

4 Power Modeling and Energy Estimation

In digital CMOS circuits, the dominant source of power consumption is switching power [22]. The average power consumed by a CMOS gate can be computed using the following equation,

$$P_{sw} = \frac{1}{2} C_L V_{dd}^2 N f,$$

where C_L is the gate output load capacitance, V_{dd} is the supply voltage, f is the clock frequency, and N is the average or expected number of output transitions per clock cycle.

Due to the influence of the switching activity on the power consumption, our main idea is to exploit the fact that power consumption is drastically reduced when some inputs of a functional unit remain unchanged for $n > 1$ clock cycles.

Here, we want to discuss the impact of the space-time mapping on the power and energy consumption respectively of the resulting processor array. Our approach identifies regions with decreased switching activity of functional units' input operands and take these power savings into account. An estimation methodology is presented in the following. This methodology estimates for a given piecewise regular algorithm and a space-time mapping T the average power consumption of the entire array.

Briefly described this methodology can be subdivided into two hierarchical estimation steps,

– PE-level power estimation,
– array-level power estimation.

4.1 PE-level Power Estimation

A sketch of a typical processor element's internal structure is shown in Fig. 2. It consists of a core part where all the functional units are located, a controller, and some delay registers. In the final paper version, we quantify the percentages

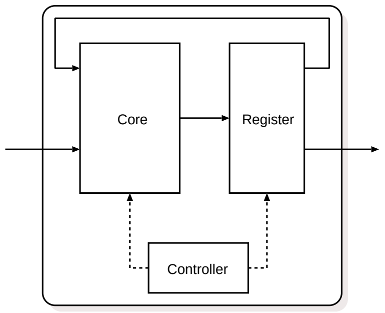


Fig. 2. Schematically internal structure of one processor element.

of power consumption for the functional units P_{FU}, the control structures P_{Ctrl}, and the registers P_{Rg} and these parts' proportion of the overall power consumption of one processing element. Then the power consumption of one PE can be approximated as follows, $P_{\mathrm{FU}}(\lambda, u) = P_{\mathrm{FU}}(u) + P_{\mathrm{Ctrl}}(\lambda) + P_{\mathrm{Rg}}(\lambda)$.

For characterization of the functional units (adders, multipliers, etc.), standard register-transfer level power estimation tools from Synopsys [23] are used.

Table 1. Average power consumption of different functional units.

n	$P_{avg,A}$	$P_{avg,B}$	$P_{avg,C}$	$P_{avg,D}$
1	26.97 μW	204.2 μW	212.0 μW	319.6 μW
2	22.33 μW	155.4 μW	164.0 μW	225.0 μW
3	18.82 μW	138.6 μW	145.6 μW	190.1 μW
4	16.99 μW	129.6 μW	137.3 μW	175.1 μW
5	16.31 μW	125.4 μW	133.8 μW	164.3 μW
6	15.68 μW	120.5 μW	128.4 μW	159.4 μW
7	15.48 μW	119.5 μW	125.2 μW	153.3 μW
8	15.29 μW	116.8 μW	124.4 μW	151.6 μW
9	15.09 μW	116.3 μW	123.7 μW	147.8 μW
10	14.89 μW	115.5 μW	122.7 μW	145.8 μW
∞_0	8.49 μW	–	–	–

In Table 1, the average power consumption of some 16-bit functional units are listed (A = ripple-carry adder, B = carry-save array multiplier, C = carry-save array multiplier with two pipeline stages, D = Wallace-tree multiplier with three pipeline stages). Each functional unit has two input operands. The value of one operand is assumed to be constant for n clock cycles; the other can change randomly in every clock cycle. These values are visualized in Fig. 3 (a) for the 16-bit ripple-carry adder and Fig. 3 (b) for the multipliers respectively. The curves are derived by regression, where the function is of type $P = a_0 + a_1 e^{-n} + a_2 n e^{-n} + a_3 n^2 e^{-n}$. The regression is good enough to have errors less than 2%. Since we are only interested in integer multiples of the clock cycle for n, the derived models may be stored in a table without too much effort.

4.2 Array-level Power Estimation

Based on the class of piecewise regular algorithms, we want to estimate the power consumption for a given space-time mapping $T = (Q\ \lambda)^{\mathrm{T}}$. It is obvious that the cost (number of processor elements) and the latency is influenced by the space-time mapping. In earlier work [11], we described how to determine the cost and the latency as a measure for performance. Here, we just briefly outline the main ideas. If we assume that processor arrays are resource-dominant, we are able to approximate the cost as being proportional to the processor count. *Ehrhart polynomials* [5, 7] may be evaluated to count the number of points (processor elements, $\#PE$) in the projected index space.

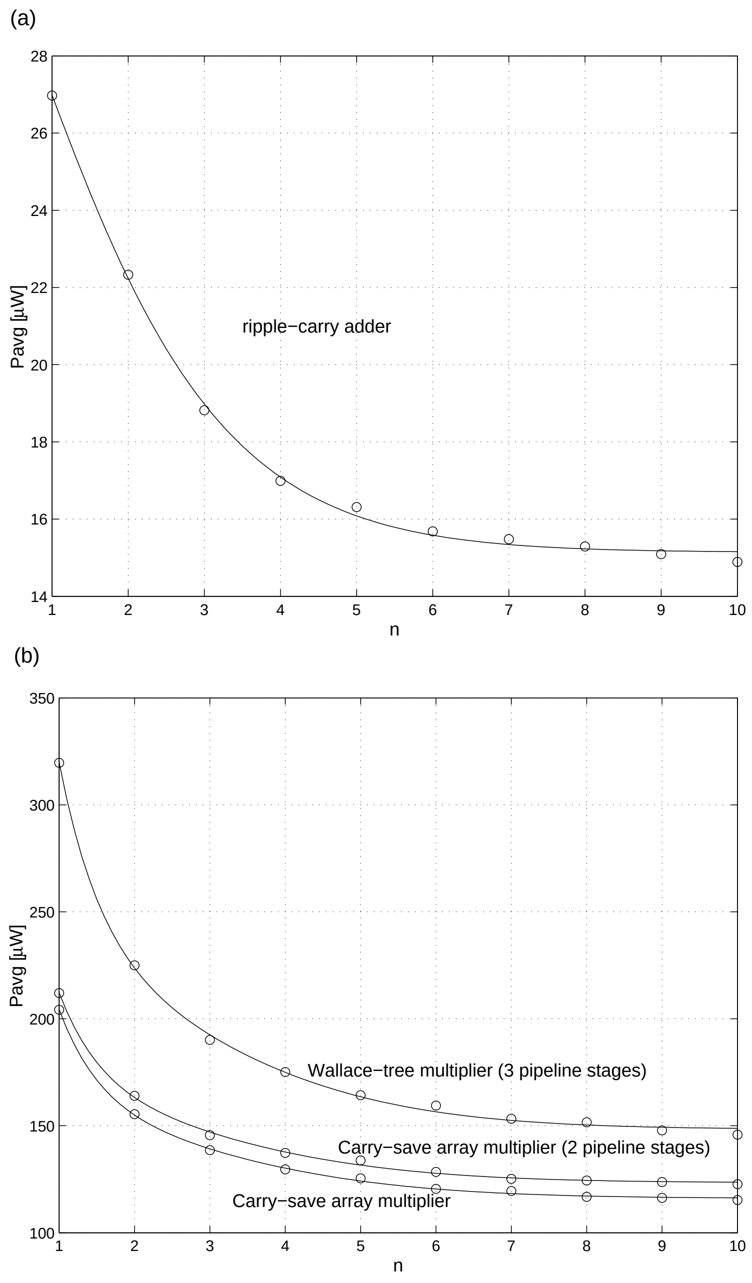


Fig. 3. Average power consumption of some 16-bit functional units when one operand is constant for n clock cycles and the other can change randomly in every clock cycle.

The latency is determined by solving a minimization problem which may be formulated as a mixed integer linear program (MILP) [25, 26]. Also, modified low power scheduling and binding techniques like in [19, 21] can be applied to compute a suited schedule.

Here, we want to discuss the impact of the space-time mapping on the power and energy consumption respectively of the resulting processor array. Our approach identifies regions with decreased switching activity of functional units' input operands and take these power savings into account. An estimation algorithm is presented on the following pages. The algorithm estimates for a given RDG G, an index space $\mathcal{I}$, a space-time mapping T, the number of processor elements $\#PE$, and the block pipelining period β the average power consumption P_{array} of the entire array. The processor count $\#PE$ and the block pipelining period β of the array may be computed as described earlier in this paper.

Once, the average power consumption P_{array} of the entire processor array is estimated, the energy consumption per problem instance is computed as follows,

$$E = \beta \cdot P_{\text{array}}.$$

Without loss of generality, we assume in the following that the iteration period π is one and that each RDG node is mapped onto a dedicated resource. Our estimation algorithm can be subdivided into two phases. In the first phase, the worst case power consumption is computed, i.e., when the switching activity of all functional units' input operands is highest. Therefore, the power consumption P_{PE} of one processor element is determined by summation of the power consumption $P_{v_i}(1)$ of all of its FUs

$$P_{\text{PE}} = \sum_{\forall v_i \in V} P_{v_i}(1).$$

The one in the term $P_{v_i}(1)$ denotes that operands can change in every clock cycle.

<u>POWER ESTIMATION</u>

```
1  IN:     RDG $G$, $\mathcal{I}$, $T = \binom{Q}{\lambda}$, $\#PE$, and $\beta$
2  OUT:    $P_{\text{array}}$
3  BEGIN
4    $P_{\text{PE}} \leftarrow 0$
5    FOR all nodes $v \in G$ DO
6        $P_{v,1} \leftarrow$ lookUpPower($v, 1$)
7        $P_{\text{PE}} \leftarrow P_{\text{PE}} + P_{v,1}$
8    ENDFOR
9    $P_{\text{array}} \leftarrow \#PE \cdot P_{\text{PE}}$
10   FOR all edges $e \in G$ DO
11     $d$ is dependence vector of edge $e$
12     node $v \leftarrow$ source($e$)
13     node $w \leftarrow$ target($e$)
```

```
      IF $(v\ =\ w)$ THEN
        IF ($S_v$ is propagation equation) THEN
          IF $(Q \cdot d = \mathbf{0})$ THEN
            FOR all adjacent edges $e'$ of $v$
              $d'$ is dependence vector of edge $e'$
              IF $(d'\ =\ \mathbf{0})$ THEN
                $w \leftarrow \mathsf{target}(e')$
                $P_{w,1} \leftarrow \mathsf{lookUpPower}(w, 1)$
                $P_{w,\beta} \leftarrow \mathsf{lookUpPower}(w, \beta)$
                $P_{\text{array}} \leftarrow P_{\text{array}} - \#PE \cdot (P_{w,1} - P_{w,\beta})$
              ENDIF
            ENDFOR
          ENDIF
        ELSE
          $(k, m) \leftarrow \mathsf{getOperandFixedCycles}(T,\ v)$
          $P_{w,1} \leftarrow \mathsf{lookUpPower}(w, 1)$
          $P_{w,k} \leftarrow \mathsf{lookUpPower}(w, k)$
          $P_{\text{array}} \leftarrow P_{\text{array}} - m \cdot (P_{w,1} - P_{w,k})$
        ENDIF
      ENDIF
    ENDFOR
  END
```

Subsequently, the power consumption of the entire array is obtained by extrapolation of this value. In the second algorithm phase array regions with lower switching activity are detected. Therefore, the whole reduced dependence graph is traversed to examine self-loops[3]. These self-loops correspond to inputs of a processor element. If these inputs remain unchanged for more than one period, the switching activity is decreased and consequently also the power. It remains to determine for how long inputs are constant and how many processor elements are affected. Two cases can be differentiated:

1. **Propagation equations mapped onto itself**. Propagation equations are only used to distribute data from one processor to another. Due to the regularity and locality of the considered processor arrays, they occur very commonly. If such a propagation equation is mapped onto itself $(Q \cdot d = \mathbf{0})$ no data transport is needed, i.e., the data remains in one processor element unchanged for β cycles until the next problem instance is fed into the array. Thus, the switching activity of all adjacent nodes v_i (functional units) in the same processor element is reduced. Therefore, the estimation of the average power consumption is decreased by $P_{v_i}(1) - P_{v_i}(\beta)$. As a propagation equation has global influence the activity is reduced in every processor element $(\#PE)$.
2. **Other self-loops**. These are the remaining inputs which may be constant for k clock cycles. The number of processor elements with these constant

[3] A self-loop is an edge where source and target node are the same.

inputs is denoted by m. Let $\mathcal{I}_{in_1}$ be the input index space of variable in_i. Transforming this index space by Q and counting the number of points in the transformed space, gives m.

$$m = \left|\left\{I \in \mathbb{Z}^{n-1} | I = Q \cdot I_{in_1} \ \wedge \ I_{in_1} \in \mathcal{I}_{in_1}\right\}\right|$$

This counting problem is similar to the earlier described one and can also solved by using Ehrhart polynomials. Once k and m are determined, the overall estimated power consumption can be reduced by $m \cdot (P_{in_1}(1) - P_{in_1}(k))$.

In the next section the overall algorithm is explained by means of discussing some results.

5 Results

Reconsider the introductory Example 1. As an allocation we choose for the addition a 16-bit ripple-carry adder and for the multiplication a three-stage pipelined Wallace-tree multiplier. The input operations a and b are mapped each to one resource of type *input*. The execution times of these operations are zero. This is equivalent to a multi-cast without delay to a set of processors. Furthermore, let $u = (1\ 0\ 0)^{\mathrm{T}}$ be the chosen projection vector. Then, after scheduling and cost calculus, we obtain the schedule vector $\lambda = (1\ 0\ 1)$ and as cost $\#PE = N_2 \cdot N_3$. Now, with this information we are able to estimate the power consumption by applying the proposed algorithm. First, the worst case power consumption is determined, i.e., the switching activity of functional units' when input operands change each both each cycle. Second, in the main part of the algorithm, two types of equations with lower input activity are detected and the overall power consumption is adapted.

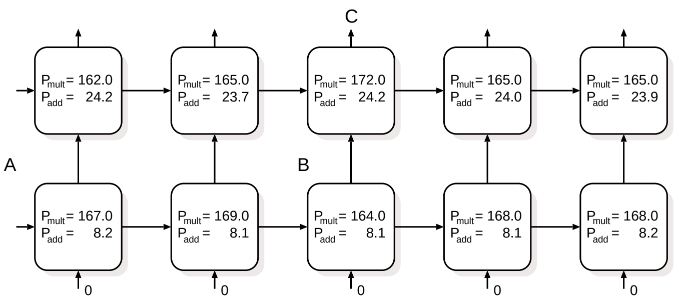


Fig. 4. Processor array for $u = (1\ 0\ 0)^{\mathrm{T}}$, $N_1 = 4$, $N_2 = 5$, and $N_3 = 2$.

The processor array for a projection in direction $u = (1\ 0\ 0)^{\mathrm{T}}$ is shown in Fig. 4. Due to this projection, the variable b is mapped onto itself. From this it follows that one operand of the multiplication remains unchanged for some time. At the beginning of a computation, the whole matrix B is input simultaneously to the array, whereas the matrix A is fed sequentially row by row from the left side into the array. Since the matrix A has N_1 rows, one operand of the multiplier is fixed for $\beta = N_1$ clock cycles which significantly reduces the power consumption in the multipliers by 45% (see Table 1). On account of the design regularity the power savings can multiplied by $\#PE$ (line 23 of the algorithm). The second point where less power is consumed is the constant input variable c. One input of the adders in the lower row of the processor array is permanently zero. This partial areas with reduced power consumption in the array are determined by the function getOperandFixedCycles. In addition to the time ($k = \infty$) where one input remains unchanged, the number $m = N_2$ of processors with reduced switching activity is returned.

Table 2. Average power and energy consumption of different mappings.

u	P_{sim} [μW]	P_{ext} [μW]	Err_{ext} [%]	P_{est} [μW]	Err_{est} [%]	E_{sim} [pJ]	E_{est} [pJ]
$(1\ 0\ 0)^{\text{T}}$	2020	3466	71.6	1928	-4.6	80.8	77.1
$(0\ 1\ 0)^{\text{T}}$	1530	2773	81.2	1456	-4.8	76.5	72.8
$(0\ 0\ 1)^{\text{T}}$	7260	6931	-4.5	6931	-4.5	145.2	138.6

In Table 2, the power consumption for different projection vectors is shown, where for illustration purposes, the upper boundaries of the index space are set to $N_1 = 4$, $N_2 = 5$, and $N_3 = 2$. In the table, P_{sim} is the exact value obtained by simulation of the entire array. The worst case extrapolation (line 4–9 in the algorithm) is denoted by P_{ext}. The power consumption of our estimation algorithm is labeled with P_{est}. Whereas the simple extrapolation method has errors up to 81%, our approach is very accurate with errors less than 5%.

Furthermore, the energy values per matrix multiplication in the table show the significant influence of the chosen space-timing mapping. Different mappings can lead to energy consumptions which differ up to a factor of two.

6 Conclusions and Future Work

A first study of a matrix multiplication algorithm has shown the great impact of a chosen mapping to the average energy consumption of the resulting array and the accuracy (errors $< 5\%$) of our estimation approach when comparing it with RTL power estimation tools from Synopsys [23].

Furthermore, our methodology is independent of the problem (array) size, since, an estimation with Synopsys design tools has linear time and memory complexity in dependence on the number of processor elements. Power estimation for large processor arrays using the Synopsys design tools rapidly becomes crucial since memory usage is growing to GBytes and estimation time to several hours. Exact comparisons of the complexity and also a quantification of the percentages of power consumption for the functional units, the controller, and the registers and these parts' proportion of the overall power consumption of one processing element are presented in the final version of this paper. First experiments of matrix multiplication and LU decomposition have shown that since all data is stored locally inside processor element's registers, the part of the register power consumption averages from $\sim 10 - 15\%$ of the overall power consumption.

Finally (in the final paper), our methodology will be verified for a piecewise regular algorithm in a case study for LU decomposition. In Fig. 5, a piecewise

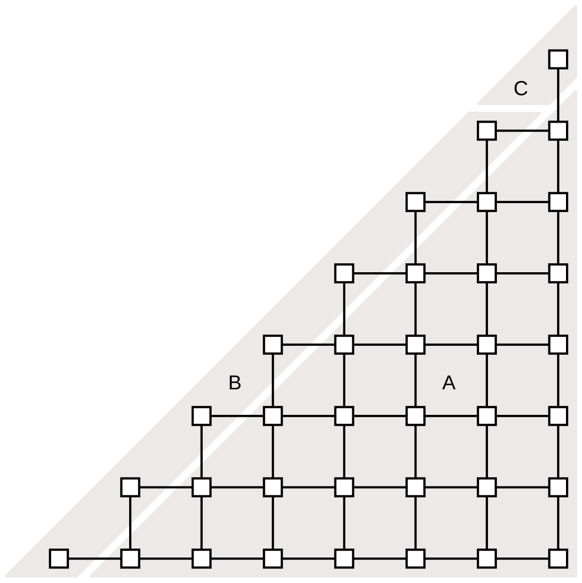


Fig. 5. Sketch of piecewise regular processor array for LU decomposition.

regular processor array for LU decomposition is schematically shown. This array can be subdivided in three pieces, where the parts A and B also change their functionality over the time.

Our new estimation methodology is currently integrated into the PARO design system and can be used during the process of automated synthesis of regular circuits. PARO is a design system project for modeling, transforming, optimization, and processor synthesis for the class of piecewise linear algorithms [1, 20].

References

1. Marcus Bednara and Jürgen Teich. Synthesis of FPGA Implementations from Loop Algorithms. In *First International Conference on Engineering of Reconfigurable*

Systems and Algorithms (ERSA'01), pages 1–7, Las Vegas, NV, June 2001.
2. Francky Catthoor, Frank Franssen, Sven Wuytack, Lode Nachtergaele, and Hugo De Man. Global Communication and Memory Optimizing Transformations for Low Power Systems. In *VLSI Signal Processing Workshop*, pages 178–187, October 1994.
3. Anantha P. Chandrakasan, Miodrag Potkonjak, Renu Mehra, Jan Rabaey, and Robert W. Brodersen. Optimizing Power Using Transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1):12–31, January 1995.
4. Jui-Ming Chang and Massoud Pedram. Module Assignment for Low Power. In *IEEE European Design Automation Conference (EuroDAC)*, pages 376–381, Geneva, Switzerland, September 1996.
5. Philippe Clauss and Vincent Loechner. Parametric Analysis of Polyhedral Iteration Spaces. *Journal of VLSI Signal Processing*, 19(2):179–194, July 1998.
6. Uwe Eckhardt. *Algorithmus-Architektur-Codesign für den Entwurf digitaler Systeme mit eingebettetem Prozessorarray und Speicherhierarchie*. PhD thesis, Technische Universität Dresden, Fakultät Elektrotechnik, Dresden, Germany, 2000.
7. Eugène Ehrhart. *Polynômes arithmétiques et Méthode des Polyèdres en Combinatoire*, volume 35 of *International Series of Numerical Mathematics*. Birkhäuser Verlag, Basel, 1. edition, 1977.
8. Paul Feautrier. Automatic Parallelization in the Polytope Model. Technical Report 8, Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex, June 1996.
9. Subodh Gupta and Farid N. Najm. Power Macro-Models for DSP Blocks with Application to High-Level Synthesis. In *IEEE International Symposium on Low Power Electronics and Design*, pages 103–105, San Diego, CA, August 1999.
10. Subodh Gupta and Farid N. Najm. Power Modeling for High-Level Power Estimation. *IEEE Transactions on Very Large Integration (VLSI) Systems*, 8(1):18–29, February 2000.
11. Frank Hannig and Jürgen Teich. Design Space Exploration for Massively Parallel Processor Arrays. In Victor Malyshkin, editor, *Parallel Computing Technologies, 6th International Conference, PaCT 2001, Proceedings*, volume 2127 of *Lecture Notes in Computer Science (LNCS)*, pages 51–65, Novosibirsk, Russia, September 2001. Springer.
12. Robert H. Kuhn. Transforming Algorithms for Single-Stage and VLSI Architectures. In *Workshop on Interconnection Networks for Parallel and Distributed Processing*, pages 11–19, West Layfaette, IN, April 1980.
13. Sun-Yuan Kung. *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.
14. Paul E. Landman and Jan M. Rabaey. Architectural Power Analysis: The Dual Bit Type Method. *IEEE Transactions on Very Large Integration (VLSI) Systems*, 3(2):173–187, June 1995.
15. Christian Lengauer. Loop Parallelization in the Polytope Model. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.
16. Diana Marculescu, Radu Marculescu, and Massoud Pedram. Information Theoretic Measures for Power Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(6):599–610, June 1996.
17. Dan I. Moldovan. On the Design of Algorithms for VLSI Systolic Arrays. In *Proceedings of the IEEE*, volume 71, pages 113–120, January 1983.

18. Enric Musoll and Jordi Cortadella. High-level Synthesis Techniques for Reducing the Activity of Functional Units. In *International Symposium on Low-Power Design*, pages 99–104, April 1995.
19. Enric Musoll and Jordi Cortadella. Scheduling and Resource Binding for Low Power. In *Int. Symp. on System Synthesis*, pages 104–109, 1995.
20. PARO Design System Project. `http://www-date.upb.de/research/paro/`.
21. Anand Raghunathan and Niraj K. Jha. An ILP Formulation for Low Power based on Minimizing Switched Capacitance during Data Path Allocation. In *IEEE Symposium on Circuits and Systems*, May 1995.
22. Anand Raghunathan, Niraj K. Jha, and Sujit Dey. *High-Level Power Analysis and Optimization*. Kluwer Academic, Norwell, Massachusetts, 1998.
23. Synopsys Inc. `http://www.synopsys.com`.
24. Jürgen Teich. *A Compiler for Application-Specific Processor Arrays*. PhD thesis, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Germany, 1993.
25. Jürgen Teich, Lothar Thiele, and Li Zhang. Scheduling of Partitioned Regular Algorithms on Processor Arrays with Constrained Resources. *Journal of VLSI Signal Processing*, 17(1):5–20, September 1997.
26. Lothar Thiele. Resource Constrained Scheduling of Uniform Algorithms. *Journal of VLSI Signal Processing*, 10:295–310, 1995.

Automatic Synthesis of Efficient Interfaces for Compiled Regular Architectures

Steven Derrien[1], Anne-Claire Guillou[1], Patrice Quinton[1], Tanguy Risset[2], and Charles Wagner[1]

[1] Irisa, Campus de Beaulieu, 35042 Rennes Cedex, France
{sderrien,aguillou,quinton,wagner}@irisa.fr
[2] LIP, ENS Lyon, 46 allée d'Italie
69364 Lyon Cedex 07
{trisset}@ens-lyon.fr

1 Introduction

The technological road map for embedded system design foresees important changes in the design methodologies. IP *based* designs and *platform based* designs [26] are becoming mandatory because the complexity of the designs increases very fast, and time to market requirements are shorter and shorter. On the other hand, improvements of technology do not affect uniformly all elements of a chip: memories, buses and power supply integration do not follow Moore's law. Hence, the main focus in hardware design has moved from high performance and parallelism to other concerns such as optimizing memory size, hierarchy and traffic, solving the bus bandwidth bottleneck, and minimizing power consumption. Power-aware design can be considered as today's most important problem, but with the emergeance of *network on chip* [7], interfacing IPs efficiently becomes a major issue. The present paper addresses this latter problem.

Many designs rely on bus protocols to interface IP, for example, AMBA or CORE-connect bus. This tendency is amplified by the spreading of the *platform based design* methodology which imposes the use of particular buses on the chip. But the gap between the design clock cycles and buses clock cycles increases, and therefore, the speed at which data can be fed into a design becomes the major bottleneck for performance. Hence, a good design must be delivered with an interface protocol that uses efficiently the bus bandwidth. Such an interface is most often designed by hand, but clearly, automatic generation of interface must be considered. This paper is mainly concerned with efficient bus protocol interface definition for highly parallel design. Note here that some attempts to plug FPGA chips directly to memories have been made [15]. In such a framework, the interface synthesis may be very different from the one presented here. However, one part of the present work, namely, how to extract interface information from the high-level specifications, remains valid, even if it has to be retargeted.

To be complete, the analysis of technological evolution should include trends in the electronic design automation tools (EDA). Currently the software technology provides satisfying tools for logic synthesis (i.e., synthesis from RTL specification.) But trends lead towards providing higher level design methodologies, using new languages (SystemC for instance) and new methodologies (use of UML specifications for instance). The research presented here belongs to a specific domain which aims at compiling loops to hardware [21, 13, 10, 2]. In this field, interfacing is also a major issue because applications operate on large data sets (usually streams in signal processing or images in multimedia) which must be efficiently brought onto the chip. Again, design automation is a major issue and the approach proposed in this paper focus on automatic interface design for architectures compiled from loop nest specifications.

Based on the work done around the MMAlpha tool [10, 8], we propose a solution to this problem for a particular class of architecture, namely linear regular arrays. In this paper, we introduce the concept of *application interface* which can be seen as the application dependent part of the interface for linear systolic arrays. As all highly pipelined designs share many properties, it is possible to define a *generic application interface*, i.e. an interface skeleton that is valid for all linear arrays and that can be easily parameterized for each application implemented. The experiments we report in this paper are oriented towards a FPGA platform, but the concepts presented can be used for interfacing IPs on a SoC, provided the IP has some features that we will describe hereafter.

The underlying interface architecture that we consider in this paper is composed of a bus (with fixed bandwidth and throughput, possibly including a faster DMA mode) and of a Fifo interconnecting the bus and the application. The Fifo allows data to be buffered when an interruption occurs at one of the bus ends. The bus and the Fifo form what we call the *hardware interface*. On top of the hardware interface is built an *application interface* whose rôle is to rearrange the data between the hardware interface and the application. This part of the interface is application dependent and can be automatically produced by the same kind of tools that generates the hardware of the application.

This paper is organized as follows. After a brief presentation in section 2 of the DLMS application which serves as an illustration throughout the paper, we explain in section 3 the model of our application architecture. Section 4 details the various elements of the interface we target. In section 5, we describe how data transfers are structured in phases and patterns to allow for efficient communications. The generation of the interface, both software and hardware, is presented in section 6. We then describe in section 7 the use of our interface generator to implement automatically a DLMS filter on a FPGA board. Finally, we present in section 8 related work and we conclude in section 9

2 The DLMS example

In this section, we introduce an example that will be used throughout to paper to illustrate our interface design: the delayed least mean square algorithm (DLMS) for channel error correction in signal processing applications.

Least mean squares adaptive filters are commonly used in signal processing applications such as echo cancellation, system identification, speech coding and channel equalization [11]. Unlike fixed coefficient Fir (Finite Impulse Response)

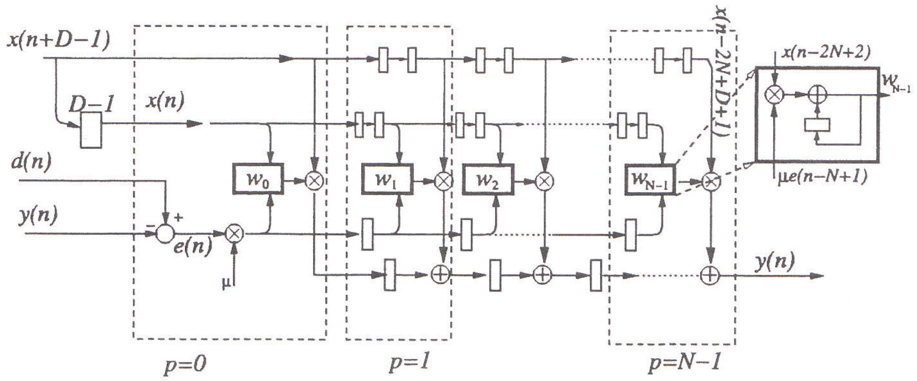


Fig. 1. Snapshot (at at $t = n - N + D$) of the architecture obtained for the DLMS after MMAlpha design process. N is the number of taps of the filter, D is the number of delays in the feedback loop. $N = 10$, $D = 12$ and $M = 100$.

or IIR (Infinite Impulse Response) digital filters, the coefficients of adaptive digital filters such as the LMS filter are adapted at each iteration to obtain better converge properties. It is well-known that recursive or adaptive digital filters are difficult to pipeline due to the presence of a feedback loop. However, it is possible to obtain a pipelined implementation by inserting delays in the recursive loop of the coefficient update part. The corresponding algorithm is called a *delayed least means square* (DLMS) algorithm.

By assuming that the adaptive digital filter is a Fir filter whose impulse response is denoted by $w_i\,(n)$, the output signal $y\,(n)$ is given by:

$$y(n) = \boldsymbol{x}^T(n)\boldsymbol{w}(n) = \sum_{i=0}^{N-1} x(n-i)w_i(n), \tag{1}$$

with $x(0) = 0$ and $\boldsymbol{w}(0) = 0$ (bold variables denotes N-vectors: $\boldsymbol{x}(n) = (x(n-N-1), \ldots, x(n))$ and $\boldsymbol{w}(n) = (w_0(n), \ldots, w_{N-1}(n))$). The weight update equations of the DLMS [12] are given by:

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \mu\; e(n-D)\boldsymbol{x}(n-D) \tag{2}$$
$$e(n) = d(n) - y(n) \tag{3}$$

where $d(n)$ is the desired signal.

A possible VLSI implementation of the DLMS [12] is represented in figure 1. In [10], it has been shown that this architecture can be derived automatically from functional specification to RTL description with the MMAlpha software, based on the Alpha language [25]. At the end of this process, the architecture presented in Fig. 1 is expressed as an AlpHard program, which is a subset of Alpha, and the mapping between the functional specification and the hardware implementation is obtained by means of the program of Fig. 2. More precisely, the

```
system firr :  {N,M,D | 3<=N<=(M-D-1,D-1)}
               (x : {n | 1<=n<=M} of integer[S,16];
                d : {n | N<=n<=M} of integer[S,16])
       returns (y : {n | N<=n<=M} of integer[S,16]);
var
  d_mirr1 : {t,p | D<=t<=-N+M; p=0} of integer[S,16];
  y_mirr1 : {t,p | D<=t<=-N+M; p=0} of integer[S,16];
  x_mirr1 : {t,p | -N+D+1<=t<=-N+M; p=0} of integer[S,16];
  x_mirr2 : {t,p | -N+2<=t<=-N+M+1; p=0} of integer[S,16];
  Y1 : {t,p | N<=t<=M; p=N-1} of integer[S,16];
let
  y_mirr1[t,p] = y[t+N-D];
  d_mirr1[t,p] = d[t+N-D];
  x_mirr2[t,p] = x[t+N-1];
  x_mirr1[t,p] = x[t+N-D];
  y[n] = Y1[n,N-1];
  use  firrModule[N,M,D] (d_mirr1, y_mirr1, x_mirr1, x_mirr2)
                 returns  (Y1) ;
tel;
```

Fig. 2. The part of the AlpHard program (`firr` system, also called AlpHard *interface*) which maps the functional specification (input x and d, result y) to the architecture (`firrModule` system also called AlpHard *module*). This system contains the information about the date and place where data should be entered (here for example, input in the first processor: $p = 0$, and output in the last processor: $p = N - 1$). This program has three parameters: `N` is the number of taps of the filter, `D` is the number of delays along the feedback loop, and `M` is the number of input samples of the filter (for simulation purposes.)

input flow `x` and the coefficient vector `d` of the initial `firr` algorithm are mapped to new variables `x_mirr1`, `x_mirr2`, and `d_mirr1`, and the architecture itself is represented by a instanciation of another Alpha program called `firrModule` (by a `use` statement), which returns an output stream `Y1`. This stream is assigned to the output variable `y` of program `firr`. All inputs and outputs of `firrModule` are indexed by `t` and `p` which represent respectively the time and the processor number to which these streams are assigned.

3 Application architecture model

This section models the interface we want to synthesize. We first state the assumptions that we make regarding the type of application hardware that we want to interface. Then we detail the information which is needed for interfacing correctly the application architecture (e.g. `firrModule` in the DLMS application) with its host architecture. Then we abstract this architecture by a number of features that will constitute the input to interface generation.

3.1 Assumptions

Our assumptions regarding the application architecture concern four aspects: a virtual clock, the model of linear array, inputs and outputs, and the bit width of streams.

Virtual clock. The application architecture is a *globally synchronous digital circuit* in which all registers are controlled by a *common virtual clock.* This virtual clock regulates the operation of the architecture, and can be frozen if, for instance, the host is not ready to send input data or to read output data. This assumption significantly reduces the control complexity inside the interface. Indeed, the designer can assume that as soon as the clock of the architecture is running, input data arrive as needed, and output data are captured by the bus when they are produced. In our experimental designs which targets FPGA chips, the virtual clock is naturally implemented using the clock enable signal of the FPGA. We also assume that the operation of the architecture begins on a `start` signal.

From now on, we call *virtual date* of a computation the number of virtual clock cycles elapsed between the computation and the start time of the algorithm.

Linear arrays. The architecture must be a linear (1-dimensional) array. Inputs and outputs are continuous streams of data, which means that all input or output streams have a virtual starting date and a virtual ending date, and no interruption occur inbetween these dates. It should be noticed that this assumption prevents from interfacing *partitioned arrays* [4], and *2-dimensional arrays.* In partitioned arrays, data arrives in *burst* mode, i.e. uninterrupted streams of data separated by long empty periods, and do not meet our assumption of continuous streams. However, the interface proposed in this paper could be easily extended to cover this case because these bursts of data are known statically [5]. In MmAlpha, architectures can be generated for any 2-dimensional regular array, but in practice, interfacing a 2-dimensional array requires a more complex architecture, since more than one data has to be provided during one virtual clock cycle.

Inputs and outputs. Any given input stream (resp. output stream) must arrive in (resp. leave) a given fixed processor, called *connection processor* of the stream. Notice that this processor may be different for each input or output stream. This assumption is most often met by the type of architecture that we are dealing with, and is easy to enforce if not.

Width of streams. We assume that the bit width of the streams is a divider of the data bus width. As the bus is usually 32 or 64 bit wide, the bit width of variables has to be a power of two (if it is not, the protocol will choose the smallest power of two greater than the actual bit width, hence reducing the efficiency of the interface.)

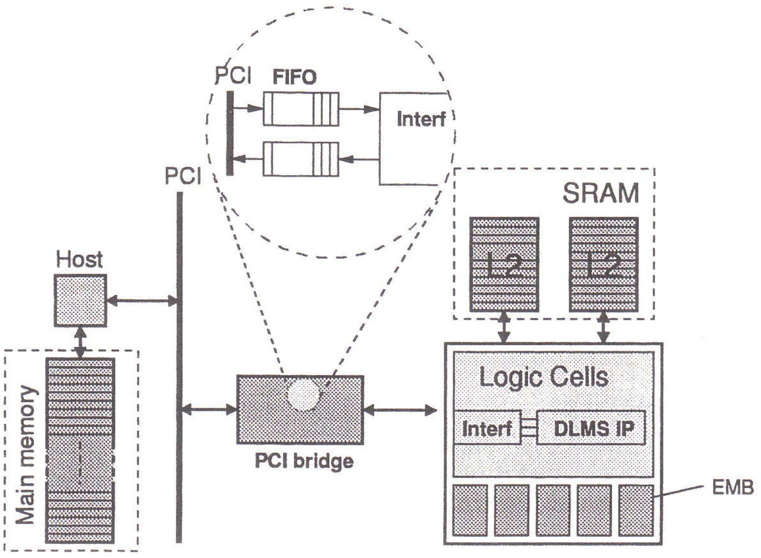


Fig. 3. Standard architecture of the interconnection between a board and the host (taken from a Spyder board [23]), together with a logical view of the Fifo mechanism used between the bus and the application architecture.

3.2 Information needed for the interface

If the above assumptions are met, then the interface of the architecture can be determined from the following information.

- The number of input and output streams. In the case of our example, the `firrModule` system has four inputs (`d_mirr1`, `y_mirr1`, `x_mirr1`, `x_mirr2`) and one output (`Y1`).
- The name, bit width, connection processor, virtual starting and stopping time of each stream. For instance input stream `x_mirr1` is 16 bit wide and is input in processor $p = 0$. The starting time is $t = -N + D + 1$ and the stopping time is $t = -N + M + 1$. All these informations can be extracted from the AlpHard interface shown in Fig. 2 (see for example the declaration of the `x_mirr1` variable in Fig. 2).

4 Interface model

In the previous section, we have described *what* we want to interface as well as the information needed to define this interface. We now describe in more details our model of interface. Fig. 3 presents a typical interface architecture, in the case of the FPGA Spyder board [23]. The application hardware, here a DLMS filter circuit, is mapped on the FPGA. The host and the DLMS are interconnected by means of a PCI bus. A PCI bridge consisting of Fifos allows for a smooth synchronization between the PCI bus and the application hardware.

In this section, we review in more details the elements of this interface: the low level interface, the application interface, and the software and hardware parts of the application interface.

4.1 Low level interface, and application interface

From now on, the interface is logically divided into two parts: the *low level interface*, and the *application interface*.

The low level interface behaves logically as a Fifo. A parameter of the low level interface is the Fifo *bit width* which we assume to be a power of 2. Notice that the implementation of the low-level interface can take different forms, depending on the target platform. In the case shown in Fig. 3, the Fifos of the low-level interface are implemented using the memory of the Spyder board. Most commercially available FPGA boards provide a similar low level interface [1, 3].

The *application interface* is the part of the interface that sends input data to (resp. gets output data from) the application IP from (resp. to) the host in order to implement a correct execution of the algorithm. The application interface is naturally divided into the *input interface* which sends data to the array, and the *output interface* which receives data from the array. These two parts are very similar, and from now on we only deal with the input interface.

4.2 Software and hardware parts

The application interface is naturally divided into a *software part* and the *hardware part*. The software part is composed of a program that sends data to the Fifo. The hardware part is more complex: it is the de-multiplexing system which gets the words out of the Fifo and sends them into the array. Of course, these two parts must be compatible, i.e. the data should be taken by the hardware part in the same order as they are produced by the software part. This is why we propose to generate the software and the hardware simultaneously from the interface AlpHard program shown in Fig. 2.

5 Structuring streams

To generate the application interface, we structure the input and output streams using two notions: *phases* and *patterns*.

5.1 Phases

A phase is a sequence of successive virtual clock ticks during which all inputs and outputs of the architecture are the same. For instance, one can see in the

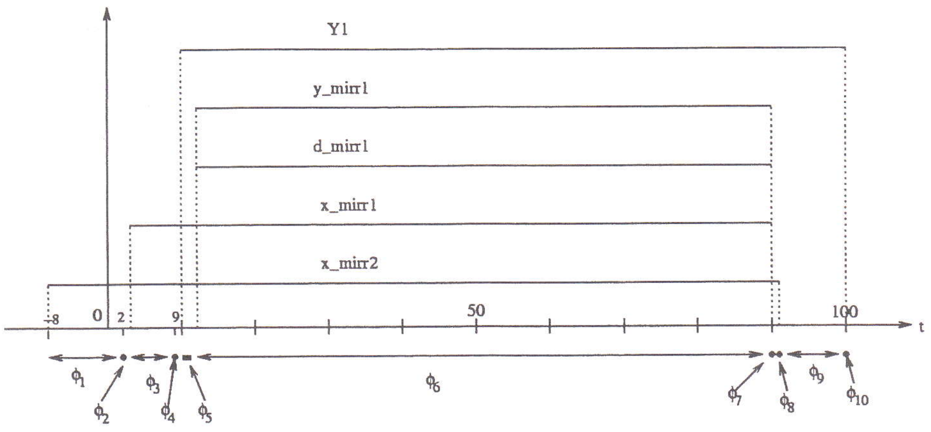


Fig. 4. The phases of the program of Fig. 2 for parameters values $N = 10$, $D = 12$ and $M = 100$

program of Fig. 2 that between clock cycles $t = -N + 2$ and $t = -N + D$, only the `x_mirr2` stream enters the array. Hence the period of time

$$\{t \mid -N + 2 \leq t \leq -N + D\} \tag{4}$$

is a phase.

During a phase, data are sent to the Fifo word by word, these words having the size of the Fifo width. For instance the `x_mirr2` variable being 16 bit wide, and if the Fifo 32 bit wide, two `x_mirr2` data can be placed in one Fifo word (remember that we assume that the bit width of all streams divides the Fifo width).

We require the length of a phase to be a multiple of the $\frac{\text{Fifo width}}{\text{data width}}$ ratio. If this is not the case, one can always break the phase in two smaller phases, in order to meet this condition. In our example (32 bit wide Fifo and 16 bit wide variable), this ratio is 2, and phase (4) contains $D - 1$ virtual clock cycles. If D is even, this interval has to be divided into two phases: phase $\phi_1 = \{t \mid -N+2 \leq t \leq -N + D - 1\}$ and phase $\phi_2 = \{t \mid t = -N + D\}$.

To illustrate this, Fig. 4 shows the phases corresponding to the program of Fig. 2, for $N = 10$, $D = 12$ and $M = 100$. We can see that $\phi_1 = \{t \mid -8 \leq t \leq 1\}$ and $\phi_2 = \{t \mid t = 2\}$.

Phases can be easily computed from the interface specification with elementary operations on the time intervals corresponding to each data. Indeed, time intervals are obtained by projecting variable domains of Fig. 2 on the time index t. In MMAlpha, these computations are done using the Polylib library [24].

5.2 Patterns

Inside each phase, a *pattern* describes in which order data are sent to the Fifo. For instance, in phase ϕ_3 of Fig. 4, one can choose to fill a Fifo word with two x_mirr1 data and then the next Fifo word with two x_mirr2 data and repeat this scheme 3 times so that 6 data of each stream are sent. In this case, the pattern is (x_mirr1,x_mirr2). One can see that this case is simple because x_mirr1 and x_mirr2 have the same bit width. In general, the choice of the pattern has to be made in order to prevent deadlock situations.

6 Generating the interface

As seen in the previous section, the *interface model* is abstracted by the description of its phases and, for each phase, its patterns. We now explain how the software and the hardware parts of the interface are generated from these informations.

6.1 Generating the software part

The software part of the application interface is the program run by the host to send data to the input Fifo or read data from the output Fifo through the PCI bus. Gererating it is far from being easy, as a set of loop nests has to be synthesized. To do this, we produce the software part as a C program by retargeting the Alpha to C compiler described elsewhere [20]. Fig. 5 shows part of the C program generated to handle inputs and outputs of phases ϕ_6, ϕ_7 and ϕ_8. Each call to WriteFifo or ReadFifo activates a function of a low level communication library.

Notice that this C program does not check whether the Fifos are ready to accept or send data: the loops of this program should therefore be *sliced* with a test on the number of slots available in the Fifo and a possible wait operation of the processor if not enough resources are available.

6.2 Generating the hardware part: principles

The hardware part is generated in VHDL and is synthesized for the FPGA chip. The architecture of the hardware part is illustrated by Fig. 6.

The hardware part is divided into the input interface and the output interface which are almost symetrical. The main difference is that the interface is started with a start signal (set up by the user, here indicating the virtual date $t = -8$), while the output interface is started by a start_out signal set up automatically (here at virtual date $t = 10$).

We detail the organisation of the input interface represented on Fig. 6. Consider an input, say input I1, ot the architecture, and let W_{I1} be the bit width of I1. This input is connected to a load32 component which is parameterized by the bit width W_{I1} of I1. The load32 component contains a shift register that allows 32 bits to be read in parallel from the Fifo and W_{I1} bits to be output during $32/W_{\text{I1}}$ clock cycles.

Each load32 component is connected to the Input_Interface component. The Input_Interface component receives data from the Fifo and store them in the appropriate load32 shift register.

```
/* phase 6 : from 12 to 89 with variables:
      {d_mirr1, y_mirr1, Y1, x_mirr1, x_mirr2}*/
for (t = 12; t <= 89; t = t + 2) {
   WriteFifo( (int *)(_d_mirr1 + (t -12)));
   WriteFifo( (int *)(_y_mirr1 + (t -12)));
   ReadFifo( (int *)(_Y1 + (t -10)));
   WriteFifo( (int *)(_x_mirr1 + (t -3)));
   WriteFifo( (int *)(_x_mirr2 + (t+8)));
}
/* phase 7 : from  90 to 90 with variables:
      {d_mirr1, y_mirr1, Y1, x_mirr1, x_mirr2} (1 data sent)*/
t = 90; {
   WriteFifo( (int *)(_d_mirr1 + (t -12)));
   WriteFifo( (int *)(_y_mirr1 + (t -12)));
   ReadFifo( (int *)(_Y1 + (t -10)));
   WriteFifo( (int *)(_x_mirr1 + (t -3)));
   WriteFifo( (int *)(_x_mirr2 + (t+8)));
}
/* phase 8 : from  91 to 91 with variables:
      {Y1, x_mirr2} (1 data sent)*/
t = 91; {
   ReadFifo( (int *)(_Y1 + (t -10)));
   WriteFifo( (int *)(_x_mirr2 + (t+8)));
}
```

Fig. 5. C code generated for phases ϕ_6, ϕ_7, ϕ_8 of Fig. 4 (for a 32 bit wide Fifo). `_d_mirr1` is an array storing value of the `d_mirr1` variable of the Alpha program.

The control of this architecture is provided by a hierarchical two level finite state machine. The states of first level are the phases of the interface, and the states of the second level are the variable names which define the patterns inside a phase. Switching from one phase to the other is done by counting the number of elapsed virtual clock cycles (for instance, we see on Fig. 4 that phase ϕ_1 must last 10 virtual clock cycles). An efficient control of the `load32` shift register allows the loading of a new Fifo word to be overlapped with the output of the last data word to the application architecture. Hence, provided that the FPGA clock frequency is high enough, the array is fed at the throughput allowed by the bus. Usually, the FPGA clock can be set up fast enough because the application design is highly pipelined but if, for instance, the input data is only 2 bit wide, the FPGA clock frequency might have to be 16 times the bus clock frequency which is probably not very realistic. All this process (and the application architecture as well) can be frozen when the data coming out of the Fifo is not ready.

6.3 Efficient synthesis of the hardware part

It is important to indicate how to efficiently implement this protocol in VHDL, as the efficiency of the final IP is greatly influenced by the efficiency of the

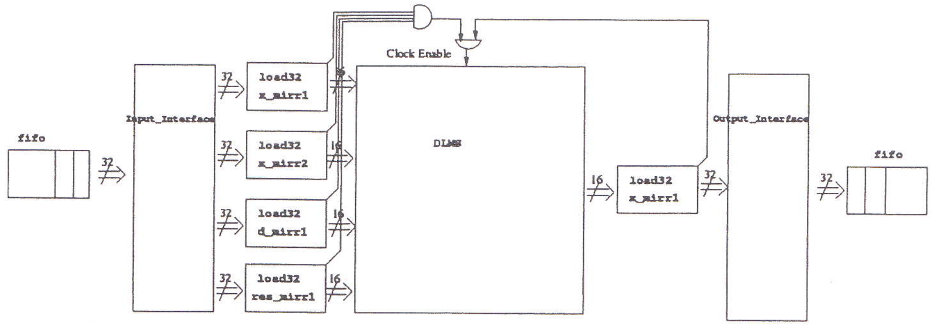


Fig. 6. Hardware part of the application interface for the DLMS

synthesis of the interface. The finite state automaton is completely generic, i.e. identical for all application architecture meeting the assumptions of section 3. The information on one phase (i.e. pattern, duration, active variables, etc) are stored in a VHDL `record`. The automaton manipulates an `array of record` whose size is the number of phases (here 10 phases.) With this implementation, on can gather all the application dependent information in one file (except very small changes like the declaration of components in the interface which depends upon the number of inputs and their bit width). This file is built by extracting information from the interface program of Fig. 2 using the same function for finding the phases and pattern as during the C code generation for the software part. Hence, compatibility between the software part and the hardware part is very easy to ensure. Part of this file is shown in Fig. 7. This coding of the automaton allows an easy automatic generation of the VHDL corresponding to the interface for any application and moreover, a very efficient implementation with commercial synthesis tools like Synopsys.

7 Experiments

The automatic interface generation was implemented and experimented for a Spyder-X2 PCI board [23], based on a Xilinx Virtex 800 device. The architecture of the board is shown in Fig. 3. In this board the host processor communicates with the FPGA through the PCI interface using memory-mapped or DMA transfers. The observed bandwidth between the host CPU and the FPGA is 8 MB/s at most.

The VHDL for the DLMS was synthesized automatically from an Alpha specification down to AlpHard (see [10].) An important issue during the design process was functional simulation. Thanks to the flexibility of the MmAlpha environment, we were able to use the same data from high level simulation down to the detailed VHDL simulation, hence speeding up the design time and allowing fast back annotation from hardware simulation to the high level synthesis process.

The application interface was generated automatically from the AlpHard interface for the DLMS algorithm (Fig. 2) for the following values of the parameters: $N = 10$, $D = 12$ and $M = 100$. The width of the interface Fifos was 32 bits.

The synthesis was realized with the `Synplify` software [22]. Table 1 gives the number of look-up tables (Lut) necessary for the synthesis of the DLMS alone, the DLMS and the interface without the Fifos, and the total design. The clock cycle, as estimated by the synthesis tool is also given. Finally, the throughput of the interface is evaluated from the cycle time and the number of data that are produced by the architecture during each cycle. More precisely, the DLMS produces one 16 bit `y_mirr` value during each cycle. The table shows also the maximum throughput of the PCI bus of the board, as observed for several designs.

One can draw some conclusions from this table.

- First, the hardware interface is not a limiting factor of speed for this design. Indeed, the clock cycle is increased only by 1ns by the Fifos.
- Secondly, the PCI bus is clearly the limiting factor of the interface: there is a factor of 8 in the best case between the bandwidth of the bus and the bandwidth that could be achieved by the design. As was expected, the design of such a high performance device is therefore limited by the communication with the host.

Note however that this interface was not optimized since the `x_mirr1` and `x_mirr2` streams are a shift in time of one another, hence only one of the two streams needs to be sent through the bus. Moreover, the `y_mirr1` stream should in practice be taken at the output of the architecture and not sent from the host (the values sent where obtained during simulations.) We choose this implementation to validate the interface protocol: indeed, the interface has a more complicated phase and pattern structure here (phases of one of two clock cycles, phases with input and outputs, etc.).

Design	Number of LUT	Clock cycle (ns)	Throughput (MB/s)
DLMS	5938 (31%)	30	66.67 MB/s
DLMS + interface	6501 (34%)	30	66.67 MB/s
DLMS + interface + Fifos	6928 (36%)	31	64.5 MB/s
PCI bus (Max)	–	–	8 MB/s

Table 1. Result of the interface generation for the DLMS algorithm. Parameters: $N = 10$, $D = 12$ and $M = 100$. This table gives the number of look-up tables occupied by the design, in the Virtex XCV800 chip (the percentage of total LUTs used in a Virtex XCV800 is given between parentheses), the clock cycle estimated by the synthesis tools, and the (one-way) throughput of the interface. The maximum observed throughput of the PCI bus of the Spyder board is given for comparison.

8 Related work and discussion

Most of the research on FPGA design focuses on the efficiency of the design itself rather than the efficiency of the interface of the design. Tests are usually made with data already in the on-board memory or with data arriving directly on board via an Analog/Digital converter. References [14, 9]. describe manually written interfaces.

In co-design oriented tools, because of the very general type of applications dealt with, communications are usually implemented by a high level synchronisation mechanism which leads to complex protocols (for instance, remote procedure call in CoWare [6].) Many FPGA compilation projects rely on control dominated models like Petri nets [27, 17], or Communicating Sequential Processes [18]. The advent of real-time operating system (RTOS) can also be a solution but efficiency will probably be degraded.

In the development of high level design tools, attempts have been made to automate interface generation. These attempts, as in the work presented here, restrict the type of architecture interfaced. In Pico [21], the interface problem is solved at run time by a bus arbiter. As the target architecture is partitioned into a small number of processors, the efficiency of the interface is not the major issue. Artemis [19, 13] relies on the Spade methodology [16] to implement efficiently communicating Kahn process networks on buses.

The methodology that we propose for the interface generation presents several novelties. First, the dynamic control is as low as possible. Indeed, it is restricted to ensuring correct behaviour when an external event such as a bus interruption occurs. Therefore, the efficiency of the interface can be statically predicted. Secondly, the design is safe because the interface is compiled *together with the architecture*, and moreover, the hardware and software parts are derived from *the same* AlpHard program and using *the same* tool. Thirdly, we have also briefly presented the simulation facilities that are available when a complete automatic design path is set with a single tool such as MMAlpha. Finally, our model is generic, and it is not only an implementation dedicated to the MMAlpha design flow but can be applied to any compiled architecture which meets the characteristics of our model.

9 Conclusion

In this paper, we have presented a tool for synthesizing interfaces for linear regular architectures. Our model of interface is bus based and contains Fifos to cope with interruptions in the flow of data arriving from the host. Our interface is application dependent, and is generated automatically from a high level description of the application, as obtained using the MMAlpha tool. Both the software part and the hardware part of the interface are generated from the same description using the same set of tools, therefore ensuring these parts to be coherent. The synthesis of the interface structures the data communications into phases and patterns, and generates a C program to be run on the host, and a VHDL program to be synthesized on the hardware platform. We have experimented this interface generator on a DLMS algorithm automatically compiled on a Spyder FPGA board and we have shown that the interface allows high performances to be reached for this application.

References

[1] Annapolis. WildStar Datasheet, 2001.

[2] F. Catthoor, K. Danckaert, C. Kulkarni, and T. Omnes. Data transfer and storage architecture issues and exploration in multimedia processors. In *Programmable Digital Signal Processors: Architecture, Programming, and Applications.* Marcel Dekker, Inc, New York, 2000.

[3] Celoxica. RC1000 Datasheet, 2001.

[4] A. Darte. Regular Partitioning for Synthesizing fixed-size systolic Arrays. *Integration, The VLSI Jounal*, 12:293–304, December 1991.

[5] A. Darte and B. Rau et F. Vivien R. Schreiber. A Constructive Solution to Juggling Problem in Systolic Array Synthesis. Technical Report 1999-15, Laboratoire de l'informatique du parallélisme, 1999.

[6] H. De Man, I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. Hardware-Software Codesign of Digital Telecommunication Systems. *Proceedings of the IEEE*, 85(3):391–418, 1997.

[7] G. De Micheli. Network on Chip: a new Paradigm for System on Chip Design. In *Design Automation and Test in Europe (DATE) 2002*, pages 418–420, Paris, 2002. IEEE Computer Society Press.

[8] S. Derrien and T. Risset. Interfacing Compiled FPGA Programs: the MMAlpha Approach. In A. Arabnia, editor, *PDPTA2000: Second International Workshop on Engineering of Reconfigurable Hardware/Software Objects.* CSREA Press, June 2000.

[9] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the Streams-C C-to-FPGA Compiler: an Applications Perspective. In *Ninth international symposium on Field programmable gate arrays*, pages 134–140. ACM Press, 2001.

[10] A.C. Guillou, P. Quinton, T. Risset, and D. Massicotte. High Level Design of Digital Filters in Mobile Communications. DATE Design Contest 2001, March 2001. Second place, available at http://www.irisa.fr/bibli/publi/pi/2001/1405/1405.html.

[11] Simon S. Haykin. *Adaptive filter theory.* Prentice-Hall information and system sciences series. Prentice-Hall, Upper Saddle River, NJ 07458, USA, third edition, 1996.

[12] M. Katsushige, N. Kiyoshi, and K. Hitoshi. Pipelined LMS Adaptative Filter Using a New Look-Ahead Transformation. *IEEE Transactions on Circuits and Systems*, 46:51–55, January 1999.

[13] B. Kienhuis, E. Rijpkema, and E.F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, 2000.

[14] D. Lavenier. SAMBA: Systolic Accelerator for Molecular Biological Application. Technical Report 988, Irisa, March 1996.

[15] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, M.Y. Wong, and K.H. Le. Pilchard - A Reconfigurable Computing Platform with Memory Slot Interface. In *Symposium on Field-Programmable Custom Computing Machines (FCCM)*, California, 2001. IEEE Computer Society Press.

[16] P. Lieverse, P. Van der Wolf, E. F. Deprettere, and K. Vissers. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3):197–207, November 2001. Special issue on SiPS'99.

[17] B. Lin and S. Vercauteren. Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation. In *international Conference on Computer-Aided Design (ICCAD)*, pages 101–109, 1994.

[18] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI signal processing*, 12:87–107, 1996.

[19] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. Van Der Wolf, and E. F. Deprettерë. Exploring embedded-systems architectures with artemis. *IEEE Computer*, 34(11):57–63, 2001.

[20] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[21] R. Schreiber, S. G. Aditya, B.R. Rau, S. Mahlke, V. Kathail, D. Cronquist, and M. Sivaraman. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing*, 2001.

[22] Synplify Pro 7.0 Reference Manual, October 2001.

[23] K. Weiß, C. Oetker, I. Katchan, T. Steckstor, and W. Rosenstiel. Power Estimation Approach for SRAM-based FPGAs. In *IEEE Symposium of Field Programmable Gate Array*, 2000.

[24] D. Wilde. A Library for doing Polyhedral Operations. Technical Report 785, Irisa, Rennes, France, 1993.

[25] D. Wilde. The Alpha language. Technical Report 827, Irisa, Rennes, France, Dec 1994.

[26] W. Wolf and N. Martinez. Session: Platform Based Design and Virtual Component Reuse. In *Design Automation and Test in Europe (DATE) 2002*, pages 296–316, Paris, 2002. IEEE Computer Society Press.

[27] X. Zhu and B. Lin. Hardware Compilation for FPGA-Based Configurable Computing Machines. In *Design Automation Conference(DAC)*, pages 697–703, New Orleans, 1999. ACM.

The Compaan Tool Chain

Realizations of the Extended Linearization Model

Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere

Leiden Embedded Research Center, Leiden
University, Leiden, The Netherlands

At the Leiden Embedded Research Center, we are working towards a framework called Compaan that automates the transformation of digital signal processing (DSP) applications to Kahn Process Networks (KPNs). These applications are written in Matlab as parameterized nested loop programs. This transformation is interesting as KPNs are well suited for mapping onto parallel architectures. Although the KPN semantic always assumes that FIFO buffers can be used between processes, we have found cases in which the FIFO is not enough as data may arrive in the wrong order. To solve this order problem, we previously presented the Extended Linearization Model (ELM) that describes a mechanism to reorder tokens. The introduction of the ELM does not violate the Kahn Process Network semantics; we still use a FIFO between a Producer and Consumer. The ELM relays on some additional memory and a controller to perform the reordering. The ELM model can be implemented in different ways. In this chapter, we investigate four different realizations of the ELM. The realizations differ in the computational complexity of performing the reordering, the kind of reordering memory used, and the size of the reordering memory.

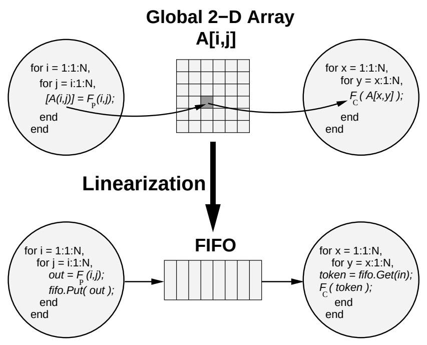
I. Introduction

An appealing and fruitful methodology to deal with exploration or designing *applications* - *architectures pairs* has become known as the *Y-chart approach*, [1]. This approach embraces two fundamental notions: the *separation of concerns* and the *abstraction hierarchy*. The concerns are: the application, the architecture, and the mapping. The abstraction hierarchy, introduced in [2] as the *abstraction pyramid*, bridges - be it for exploration or synthesis purposes - the gap between high level application specification and low-level architecture specification by defining a number of abstraction levels and a corresponding *stack of Y-charts*. At each level, application models, architecture models, and mapping models must match to make exploration and synthesis feasible.

Several research groups around the globe are currently experimenting with this methodology, some explicitly and others implicitly. They are, naturally, all focusing on different application domains which lead to different views on this methodology. Applications in the realm of *automotive*, *multimedia*, and *communications* have different requirements, constraints, and boundary conditions which result in different challenges.

The Leiden Embedded Research Group focuses on applications that can be specified as parameterized affine Nested Loop Programs (NLPs). The group has been developing and implementing the *Compaan* tool chain to translate such applications from their imperative language specification into Kahn Process Networks (KPN) [3]. The application specification language is Matlab or C, and the tool-chain is a compiler through which a range of KPNs can be obtained for any given application specified as a parameterized NLP.

The processes in the Compaan generated KPNs are not (completely) specified in an imperative model of computation because the distance between that model and the models in which architecture components - in particular the processing units - are specified is too large. This is not specific to the application domain for which Compaan is an appropriate translation tool set; it is a problem that is revealed wherever the Y-chart methodology is used. Of course, the processes in KPNs may be specified in terms of more than one model of computation. For example, one could be obtained for the *Control Data Flow Graphs* model [4] or for one or more



```
for i = 1:1:N,
  for j = i:1:N,
    [A(i,j)] = F_P(i,j);
  end
end
```

```
for x = 1:1:N,
    for y = x:1:N,
      F_C( A[x,y] );
    end
end
```

```
for i = 1:1:N,
    for j = i:1:N,
        out = F_P(i,j);
        fifo.Put( out );
    end
end
```

```
for x = 1:1:N,
    for y = x:1:N,
token = fifo.Get(in);
F_C( token );
     end
end
```

Figure 1. The Standard Linearization Model

Dataflow Network models [5].

The Process Network Model (PN) in Compaan is the *Kahn Process Network (KPN)* model [6], which consists of concurrent autonomous processes that communicate in a point to point fashion over unbounded FIFO channels using a blocking-read synchronization. The strength of a Process Network is that it uses no global memory and no global scheduler. This makes a KPN very appealing for further implementation into hardware [7].

In the Compaan KPN processes, each process executes an internal function following a local schedule. At each execution (also referred to as *iteration*) this function reads/writes data from/to different FIFOs. An input port domain (IPD) of a process is the union of the iterations at which the process's function reads data from the same FIFO. An output port domain (OPD) of a process is the union of the iterations at which the process's function writes data to the same FIFO. Each FIFO uniquely relates an input port to an output port forming to an instance of the classical *Producer/Consumer* pair [8].

One of the tools in the Compaan Tool Chain is *Panda*. Panda accepts as input the description of a Polyhedron Reduce Dependence Graph (PRDG) and transforms this PRDG into a Process Network (PN). This transformation is done in a number of steps. One of the steps involved is the *Linearization*, in which a high dimensional data structure (e.g., matrix $A[i,j]$) is linearized into a single linear stream of data. In case of Kahn process networks, the linearization model is a FIFO buffer as shown in Figure 1.

In the top part of this figure, a producer and consumer process are given that communicate the data array `A[i,j]` using global memory. In the linearization step, this communication is replaced with a FIFO buffer, leading to the producer/consumer processes given in the lower part of Figure 1. Observe that in the top part, the indices *i* and *j* are used to address matrix A. In the bottom part, the reference to A has been eliminated. The for-loops only describe an order and data produced by the function is placed on the FIFO buffer. There are cases, however, in which a FIFO as the Linearization Model (LM) no longer holds. If the order data is produced is different from the order data needs to be consumed, a FIFO buffer no longer is enough. In [9], we have proposed an extension to the LM, which we called the *Extended Linearization Model* (ELM). This model includes an additional reordering mechanism that consists of a *Controller* unit and some *Reordering Memory*. The ELM preserves the semantics of the KPN model. As we will show in this chapter, the ELM can be realized in different ways and each realization has its own strength and weakness. Based on these realizations, alternative hardware/software mappings of the Compaan generated network onto different platforms are feasible.

II. In Order/Out of Order case:

Consider the two KPN processes in Figure 2 with node domains `P` = $\{$ p, I $\}$ and `C` = $\{$c, $K\}$, respectively that are collections of atomic nodes `p`(i,j) and `c`(x,y) defined on the domains $I = \{(i,j)|\ 3 \leq j \leq N \wedge 1 \leq i \leq N-2\}$ and $K = \{\ (x,y)|\ 2 \leq x \leq N-1 \wedge 2 \leq y \leq N-1\ \}$, respectively. In the first process one of the OPDs is `O` = $\{$ `out`, J $\}$ that is a collection of atomic output ports `out`(i,j) defined on the domain $J = \{(i,j)|\ 3 \leq j \leq N \wedge i+2 \leq j \wedge 1 \leq i \leq N-2\}$. In the second process one of the IPDs is `I` = $\{$ `in`, L $\}$ that is a collection of atomic input ports `in`(x,y) defined on the domain $L = \{(x,y)|\ 2 \leq x \leq N \wedge x \leq y \wedge 2 \leq y \leq N\}$. There is a mapping M $(i = x-1; j = y+1;)$ relating these two port domains. Hence, these two ports form a Producer/Consumer pair. A token produced by the atomic node `p`(i,j) is put on the FIFO channel reserved for the edge domain (`O`, `I`) through the atomic output port `out`(i,j), and will be consumed by the atomic node `c`(x,y) through the atomic input

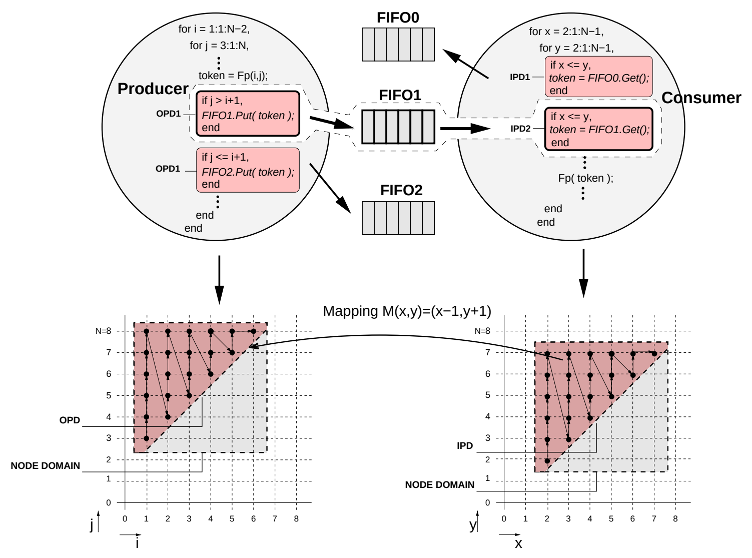


Figure 2. A Producer and Consumer process. Of the Producer we show the output port domains (OPDs) and of the Consumer, we show the input port domains (IPDs). Each OPD is uniquely connected to another IPD via a FIFO. Over this FIFO, tokens are communicated that adhere to the mapping given by the mapping matrix M. In this example, OPD1 is connected to IPD2 via FIFO1. The Producer/Consumer with the FIFO form an instance of the classical consumer/producer pair.

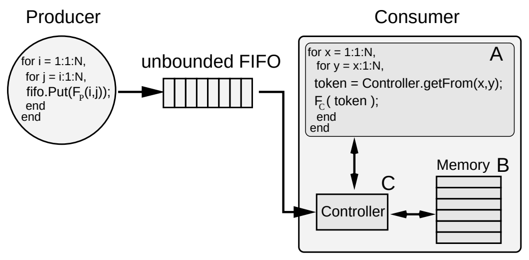
port `in`(i+1, j-1) that gets the token from this channel.

Since the KPN processes are sequential processes, no two atomic ports in a port domain are active at the same time. That is, there is an order among the atomic output ports in an output port domain, and there is an order among the atomic input ports in the corresponding input port domain. In [9], we have defined the *rank* function that expresses in a pseudo-polynomial form this order of execution in a particular domain. The rank function is derived using the Ehrhart theory that expresses the number of integral points inside of a polytope as a pseudo-polynomial expression [10]. A pseudo-polynomial is a polynomial with periodic coefficients. This theory has been extended recently for parameterized polytopes [11].

As a consequence, the expression of the rank function is in general a set of pseudo-polynomial expressions depending on the parameters. Examples of the rank functions will be shown later when various realizations of the ELM are discussed.

In Compaan, the sequential ordering of atomic nodes firing in a node domain is in *lexicographical order*, which means these nodes are scheduled according to a loop nest. The ordering in which tokens are put on a channel is the same as the order in which atomic nodes are fired. Because the channel is a FIFO channel, a consumer can only get the tokens from the channel in the same order. This represents the *in-order* case. However, depending on the lexicographical schedule of the consumer's atomic nodes, the consumption of the channel tokens may follow a different order than the order in which these tokens were put on the channel. This represents the *out-of-order* case. To work correctly in the out-of-order case, a Consumer needs a mechanism to restore the consumption order. This mechanism relays on the use of private *reordering memory* for temporary storage of tokens. Once stored, the tokens can be consumed in the correct order. This reorder mechanism is modeled as the ELM.

III. The Extended Linearization Model in more detail



```
for i = 1:1:N,
  for j = i:1:N,
  fifo.Put(F_P(i,j));
  end
end
```

```
for x = 1:1:N,
  for y = x:1:N,
  token = Controller.getFrom(x,y);
  F_C( token );
  end
end
```

Figure 3. The Extended Linearization Model

The main elements in the Extended Linearization Model are the local reordering memory and the Controller. Because the tokens can no longer be read directly from the FIFO, as they may arrive in the wrong order, they are delivered by the Controller to the function unit. In this way, the

Controller takes care of supplying tokens to the consumer Function in the right order. In Figure 3, a schematic representation is given of the ELM. It shows the Consumer process (A), the Reorder Memory (B), and the Controller (C).

A. The Process Description (A)

The process description in the ELM is different from the process description when using the LM. Instead of getting tokens directly from a FIFO, the function gets its tokens from the Controller. Hence function call *fifo.Get* (See the lower part of Figure 1) is replaced with the call to the Controller function *getFrom*.

B. The Memory (B)

The Memory stores tokens allowing the Controller to reorder tokens into the order required by the Consumer process. Two kinds of memory are possible: Random Access Memory (RAM) and Content Addressable Memory (CAM). The two kinds of memory differ in the way they are addressed. The implementation of the Controller depends on the type of the memory.

C. The Controller (C)

The Controller converts the sequence tokens are produced into the sequence they have to be consumed. The Controller performs this reordering by addressing the reordering memory (B). This functionality is exposed externally to the Consumer process by the function *getFrom(x,y)* that returns the token to function F_C for an arbitrary iteration point (x, y).

The behavior of the Controller is shown in pseudo code in Figure 4. The `getReadAddress(x,y)` determines the memory address of the token needed at the iteration (x, y). Next, the Controller checks whether the token is already available at that address by calling the function `emptyMem`. If the token is present, it is read from that address by calling the

```
Token t getFrom(x,y) {
   double address = getReadAddress(x,y);
   if( ! emptyMem(address) ) {
     return readFromMem(address);
   } else {
     return readFromFifo(address);
   }
}
```

Figure 4. The components in the Controller.

function `readFromMem`. Otherwise, the Controller starts to read tokens from the FIFO and stores them in the memory until the desired token arrives at the address of interest. The procedure of reading from FIFO is initiated using the function call `readFromFifo`. Storing tokens into the memory implies that for each token read from the FIFO, a certain address is generated. Depending on the type of memory used, different procedures are available to generate this address. These procedures are realized as the function *getWriteAddress* inside the function `readFromFifo` .

IV. Realizations of the Extended Linearization Model

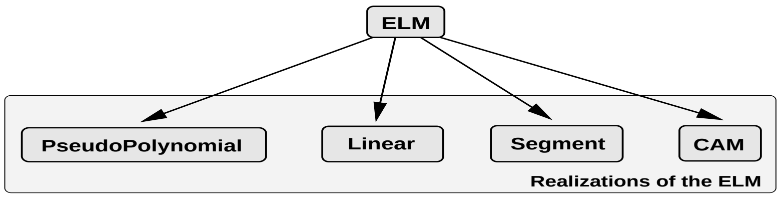


Figure 5. Four Model Instances

The ELM can be realized in four different ways as shown in Figure 5. The realizations differ by the way the function *getReadAddress* and function *getWriteAddress* are implemented and by the type of memory used as reordering memory. To compare the four different realizations, the following three characteristics are relevant:

The complexity of the addressing mechanism the computational complexity of the controller functions *getReadAddress* and *getWriteAddress*.

The dimension of the reordering memory the number of the storage locations needed to perform the reordering.

The generality of the realization the class of algorithms for which Compaan can derive KPNs.

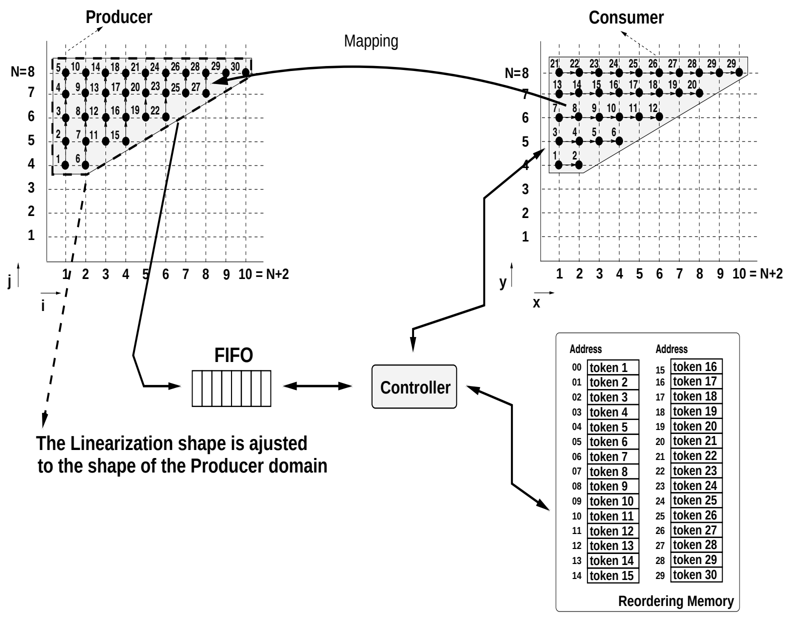
To introduce the four realization, we use as an example the Producer/Consumer pair given in Figure 6. The graphical representation of the domain descriptions of the Producer/Consumer pair is shown in the top part of Figure 7. Because the order the Producer produces data is different from the order the Consumer consumes, an ELM realization is needed in the linearization of the Producer/Consumer pair.

```
for (int i=1;i<=N+2;i++){          for (int y=4;y<= N;y++){
   for (int j=1;j<= N;i++){           for (int x=1;x<= N+2;x++){
      if (2*j >= i+6){                   if (x <= 2*y-6){
          a[i,j] = Fp();                     Fc(a[x,y]);
      }                                  }
   }                                  }
}          Producer                }          Consumer
```

Figure 6. Running Example

V. PseudoPolynomial realization

The PseudoPolynomial realization is based on the fact that the order of the iterations inside an OPD can be expressed as a pseudo-polynomial, which is the rank function discussed earlier in this chapter. In general, the *getReadAddress* function of the Controller is a pseudo-polynomial function. In Figure 7, the iteration points of the OPD are perfectly enclosed by a shape that we call the *linearization shape*. The pseudo-polynomial expression is computed by calculating the $rank$ function inside the Linearization shape. This consists of adding several pseudo-polynomials P_i,


Producer
Mapping
Consumer
N=8
j
i
1 2 3 4 5 6 7 8 9 10 = N+2
y
x
FIFO
Controller
The Linearization shape is ajusted
to the shape of the Producer domain
Reordering Memory

Address		Address	
00	token 1	15	token 16
01	token 2	16	token 17
02	token 3	17	token 18
03	token 4	18	token 19
04	token 5	19	token 20
05	token 6	20	token 21
06	token 7	21	token 22
07	token 8	22	token 23
08	token 9	23	token 24
09	token 10	24	token 25
10	token 11	25	token 26
11	token 12	26	token 27
12	token 13	27	token 28
13	token 14	28	token 29
14	token 15	29	token 30

Figure 7. The PseudoPolynomial realization.

$P_2, ... P_n$, where n is equal to the dimension of the Linearization shape. For our running example, the rank is the sum of the two pseudo-polynomials P_1 and P_2:

$$\begin{aligned} rank(i,j) &= P_1(i,j) + P_2(i,j) \\ P_1(i,j) &= (i-1)*N + (-1/4)*i^2 - 2*i + [2, 9/4]_i \\ P_2(i,j) &= j + (-1/2)*i + [-3, -7/2]_i \\ rank(i,j) &= -1/4*i^2 + (N-5/2)*i + j + [-1, -5/4]_i - N. \end{aligned} \tag{1}$$

To obtain the *getReadAddress* function, the rank needs to be composed with the mapping $M(x,y)$ and the result is equal to :

$$\begin{aligned} getReadAddress(x,y) &= rank(i,j) \circ M(x,y) \\ &= -1/4*x^2 + (N-5/2)*x + y \\ &\quad - [1, 5/4]_x - N. \end{aligned} \tag{2}$$

The $rank$ polynomial contains all the information needed by the Consumer process to reorder the token correctly. For this realization, tokens are written into the reorder memory following the sequence into which they arrive from the FIFO. Therefore, the function *getWriteAddress* is a simple increment. The dimension of the reordering memory is equal to the number of iteration points in the OPD. For the Producer/Consumer of Figure 6, the dimension is equal to $(N-4)*(N+2)/2$. The computational complexity of addressing the reordering memory can be quite large. It requires the evaluation of a pseudo-polynomial expression like, for example, the one given in equation 2. In general, the PseudoPolynomial realization is valid only for the cases when an OPD is a polytope. Under certain conditions, the realization can be extended for cases an OPD is not a polytope [12].

VI. Linear realization

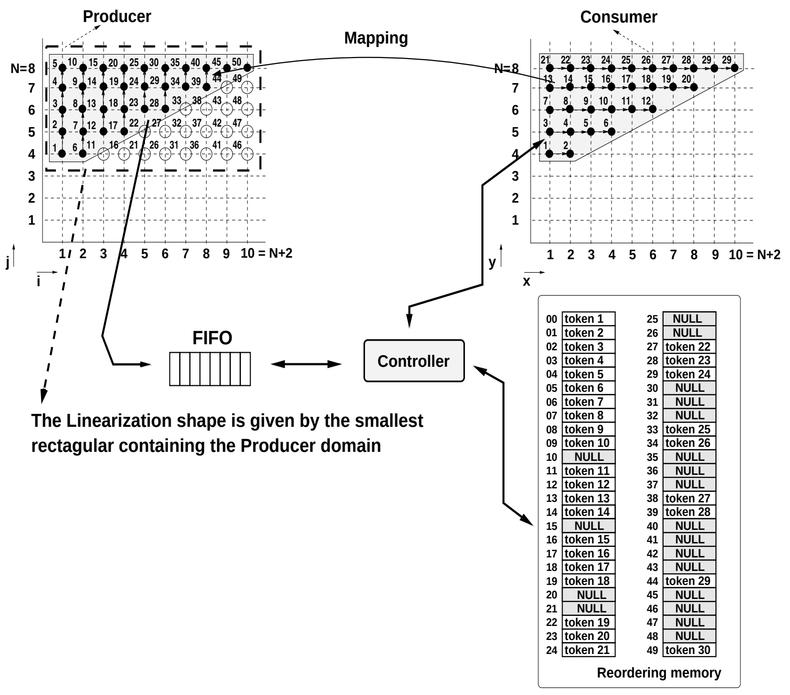
The Linear realization is based on the classical Linearization of an n-dimensional array into a one-dimensional array [13, 14]. The classical Linearization shows that a rectangular shape can be addressed using a simple polynomial. Inspired by this concept, we relax the Linearization shape to the smallest rectangular that includes the producer domain (OPD). Consequently, the *getReadAddress* that results is always a simple linear function. The rectangular Linearization shape is shown in Figure 8, and the $rank$ function is as follows:

$$rank(i,j) = (N-3)*(i-1)+j-4. \tag{3}$$

The *getReadAddress* function is obtained by composing the rank function with the mapping function $M(x,y)$, and the final polynomial expression is :

$$\begin{aligned} getReadAddress(x,y) &= rank_P(i,j) \circ M(x,y) \\ &= (N-3)*(x-1)+y-4. \end{aligned} \tag{4}$$

The consecutive order inside the Linearization shape, however, can get disturbed. This happens when an OPD doesn't have a rectangular shape and therefore, more iteration points are enclosed by the Linearization shape than necessary. As a consequence, these additional iteration points are also


Producer
Mapping
Consumer
N=8
j
i
10 = N+2
y
x
FIFO
Controller
The Linearization shape is given by the smallest rectagular containing the Producer domain
Reordering memory

00	token 1	25	NULL
01	token 2	26	NULL
02	token 3	27	token 22
03	token 4	28	token 23
04	token 5	29	token 24
05	token 6	30	NULL
06	token 7	31	NULL
07	token 8	32	NULL
08	token 9	33	token 25
09	token 10	34	token 26
10	NULL	35	NULL
11	token 11	36	NULL
12	token 12	37	NULL
13	token 13	38	token 27
14	token 14	39	token 28
15	NULL	40	NULL
16	token 15	41	NULL
17	token 16	42	NULL
18	token 17	43	NULL
19	token 18	44	token 29
20	NULL	45	NULL
21	NULL	46	NULL
22	token 19	47	NULL
23	token 20	48	NULL
24	token 21	49	token 30

Figure 8. The Linear realization

ranked by the $rank$ function, disturbing the consecutive order. Looking at Figure 8, we see that after iteration 10 follows iteration 11. However, iteration 11 does not belong to the OPD. The next iteration belonging to the OPD has rank 12 and hence the order becomes 10, 11, 12. Consequently, the Controller cannot rely any longer exclusively on the order tokens are read from the FIFO; the eleventh token read from the FIFO should be written at the address 12. Therefore, the Controller cannot use a simple increment for the *getWriteAddress* function.

To re-create the correct sequence of addresses, the Controller relays on a function that assigns to incoming tokens the correct order number inside the OPD. This function is called the *recover* function. This function re-implements at the Consumer side the logic used to schedule the iteration points inside the OPD.

The advantage of this realization, is that the function used to address the reordering memory is always a linear expression depending on the coordinates of the consumer iteration point. A disadvantage is that need for the *recover* function. Moreover, the extra iteration points enclosed by the linearization shape result in empty memory slots (represented by the “Nulls” in the memory in Figure 8). In the example, the memory requirement is equal to the dimension of the Linearization shape, i.e., to $(N-3)*(N+2)$, but only half of this space is actually used.

VII. Segment realization

The PseudoPolynomial realization results in good memory usage, but the addressing formula can be very complex because of the irregularities it contains as expressed by the periodic coefficients. On the other hand, the Linear realization results in simple addressing but potentially wastes a lot of memory. We now present the Segment realization that combine the best features of the two approaches discussed so far: simple addressing mechanism and efficient memory usage.

The Segment realization is based on the fact that pseudo-polynomials can be decomposed into a *linear part* and a *non-linear part* as shown in Figure 10. The linear part describes the consecutive order, the non-linear part described the non consecutive order. At the Producer side, the order changes at iteration points at which the innermost nested loops start to iterate again from their lower bound value. We say that a *non-linearity* occurred at iteration point (IP) and using the notion of these IPs, the Segment realization computes the value of the pseudo-polynomial using a *Segment Number* and a *Segment Displacement* as is shown in Figure 10. How the Segment Number and Displacement are computed is explain later on in the chapter. Because the segment number and displacement are pre-computed, a pseudo polynomial cannot be evaluated in a parameterized way, as is possible in the PseudoPolynomial realization. Hence, parameter values in a NLP have to be fixed in order to use the Segment realization.

Writing data into the reordering memory occurs in the same way as in the PseudoPolynomial realization. The Controller writes tokens in the *Reordering Memory* as they arrive from the FIFO. The detection of the IPs at which the consecutive order get disturbed, is done by the *recover*

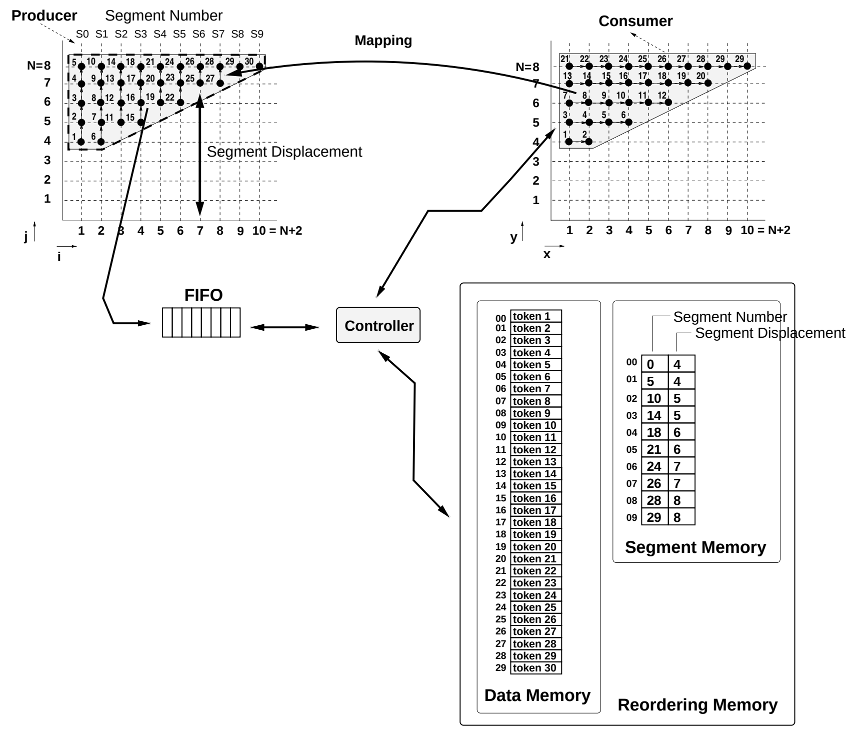


Figure 9. The Segment realization

function that duplicates the Producer for-loops at the Consumer side. With each occurrence of an IP, a *Segment Number* and *Segment Displacement* is associated. Each such number pair is used by the Controller to determine the value of the Pseudo polynomial. Let's see how writing and reading takes place in the Segment realization. The writing is implemented in the *getWriteAddress* and the reading is implemented in the *getReadAddress*

Writing a token happens in the following way. Initially, the Controller contains an internal counter that is set to zero. The *recover* function keeps track of whether the order is linear or non-linear. if the order is linear, a token is read from the FIFO, and the internal counter is incremented by one. If the order is non-linear, the Controller allocates a new entry in the *Segment Memory*. It writes in the entry the current value of the iterator in the *Segment Displacement* field and the currently value of the counter in

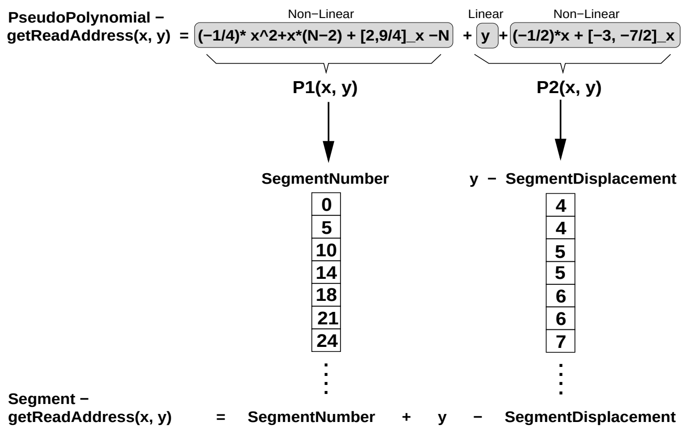

PseudoPolynomial – getReadAddress(x, y) = (−1/4)* x^2+x*(N−2) + [2,9/4]_x −N + y + (−1/2)*x + [−3, −7/2]_x
Non−Linear
Linear
Non−Linear
P1(x, y)
P2(x, y)
SegmentNumber
0
5
10
14
18
21
24
y − SegmentDisplacement
4
4
5
5
6
6
7
Segment – getReadAddress(x, y) = SegmentNumber + y − SegmentDisplacement

Figure 10. Computation of a Pseudo Polynomial using a Segment Number and a Segment displacement stored in the Segment Memory.

the *Segment Number* field.

In Figure 9, the Producer starts at iteration $(1, 4)$, which immediately results in an IP for iterator i. Consequently, an entry is allocated in the segment memory at address 0. The counter has a value of 0 and the iterator is equal to 4, leading to entry (0,4) at address 0. Next, iterator i moves consecutive to iteration $(1, 8)$. At the next iteration of i, an IP occurs again. A new entry is generated at address 1. The counter value is equal to 5 and the value of i is again 4, leading to the (5,4) entry at address 1 and so one.

For a particular iteration point of the Consumer, the Controller determines the address from where data has to be consumed using a three-step procedure. The three steps are:

$$\begin{aligned} &\text{step1:} \quad (i, j) = Map \circ (x, y). \\ &\text{step2:} \quad Segment = SegmentMemory(i - i_{start}). \qquad (5) \\ &\text{step3:} \quad address = Segment_{Number} + j - Segment_{Displacement}. \end{aligned}$$

In Figure 9, the Producer starts from the iteration (i_{start}, j_{start}), which is equal to $(1, 4)$. Suppose the Consumer wants to obtain the token for iteration $(4, 6)$. In step 1, the iteration is mapped in an iteration at the Producer and is equal to $(4, 6)$ (i.e., the mapping is identity). In step 2, the segment is found associated with this iteration. In step 2, i_{start} is equal to 1 and i is equal to 4. Hence, the segment number is 3, which is address 3 in the segment memory were entry $(14, 5)$ is stored. In step 3, the address in the reordering memory is calculated as 14 + 6 - 5 = 15. At address 15, the token is stored that was generated at the 16th iteration by the Producer. This is the token needed by iteration $(4, 6)$ of the Consumer as can be verified by inspection in Figure 9. The memory size of this realization is equal with the size of the DataMemory plus the size of the SegmentMemory. In our example the size of the DataMemory is $N^2/2 + 1/2N - 6$ (the same as the memory size from the PseudoPolynomial realization) and the size of the SegmentMemory is $N + 2$. Hence, the total memory size is equal to $N^2/2 + 3/2N - 4$.

VIII. CAM realization

The CAM realization uses a Content Addressable Memory (CAM) as reordering memory. In a CAM, a *key* is used instead of an address to access the content of the memory. The entry used in the CAM realization is given in Figure 11. It shows that each entry in the CAM consists of a key, the token associated with the key, and a field called *multiplicity*. We explain later what the term multiplicity means. The CAM approach works, because to each token produced at the producer OPD an *unique* key can be associated. The Controller can reproduce this key to obtain the token the Consumer requires at a particular iteration.

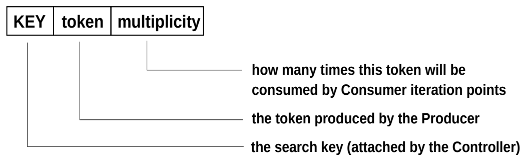

KEY
token
multiplicity
how many times this token will be consumed by Consumer iteration points
the token produced by the Producer
the search key (attached by the Controller)

Figure 11. The CAM entry

For the CAM realization, the function *getReadAddress* generates a key instead of an address. The generation of the unique key can be done in different ways. We compute the $rank$ function inside the Producer based on an Node domain instead of an OPD. Another possibility would have been to use a classical linearization polynomial. In general, the shape of a Node domain results in a simple polynomial instead of a pseudo-polynomial and it therefore easily calculated. Using the $rank$, a unique number is associated that is equal to the order of the iteration inside the Node domain of the Producer. To illustrate this, consider again the Producer/Consumer pair from Figure 6. Suppose that the Node domain is defined as

$$C = \{(i,j) |\ 1 \leq i \leq N \wedge 1 \leq j \leq N+2\},$$

Then the $rank$ is given by

$$rank(i,j) = i * N + j.$$

By composing the $rank$ with the mapping $M(x,y)$ we obtain the *getReadAddress* function:

$$getReadAddress(x,y) = x * N + y. \tag{6}$$

For a given iteration point (x,y) of the Consumer process, the *getReadAddress* function determines the unique key for that iteration using equation 6. For this key, the Controller checks (using function *EmptyMem*) if a token already exists in the CAM by searching all keys for a match. If no match can be found, the token is not stored yet.

If the key exists, the token associated with the key is retrieved from the CAM by function *readFromMem*. If the token doesn't exist, the Controller keeps loading data from the FIFO into the CAM. This happens in function *ReadFromFifo*. To each token the Controller loads, it attaches an unique key given by the function *getWriteAddress* and multiplicity number. Loading data from FIFO stops upon arrival of the token for which the key (as given by *getWriteAddress*) is the same as the key the Controller is searching for (as given by *getReadAddress*). The function *getWriteAddress* is based on a $recover$ function similarly to the recover function from the Linear realization.

In general, a token is read only once by the Consumer process. There are cases in which the same token is read more than once by the Consumer process. This called a *broadcast*. A read from a FIFO is destructive and in case of a broadcast, this would mean that a token needs to be send over the FIFO as many times as needed, or that a token needs to be stored in memory and read from memory as many times as needed. In the CAM realization, we implemented the latter option as it is more efficient.

To keep track of how many times a token is to be read, we have introduce the notion of *multiplicity* [12], which indicates how many times a particular token needs to be read by the Consumer process. Each time a token is consumed, its multiplicity is decremented. When the multiplicity reaches zero, no other iteration will need that token and it can be erased. That location can be reused by other tokens. Hence, using multiplicity, the Controller is able to free memory locations and consequently, this realization uses the smallest possible amount of memory. The memory size (MS) of the CAM is given by the next formula:

$$\text{MS} = \max_{(x,y)\in C}(read(x,y) - rank_{Consumer}(x,y)) + 1. \qquad (7)$$

where C represents a sub-domain of the Consumer domain where no two points read the same token. In the case from Figure 6, C is the whole Consumer domain. The $read$ function is the same as the *getReadAddress* function from the PseudoPolynomial realization:

$$read(x,y) = -1/4 * x^2 + (N - 5/2) * x + y - [1, 5/4]_x - N. \qquad (8)$$

and the $rank_{Consumer}$ is the function that gives the order of the Consumer iteration points:

$$rank_{Consumer}(x,y) = y^2 - 7y + x + 11. \qquad (9)$$

According to equation 8 and 9 it resultes that:

$$\text{MS} = \max_{(x,y)\in C}(-1/4 * x^2 + (N - 7/2)x - y^2 + 8y - N - [11, 45/4]_x).$$

The maximum of this formula inside the C domain can be derived using analitycal methods. In our case for $N = 8$, we have $MS = 10$. For more informations about the read and the rank function, we refer to [9]. The key to efficient memory usage is the ability to compute the multiplicity for a token. However, this multiplicity is again computed using the Ehrhart theory and may again be a pseudo-polynomial.

IX. Comparing the different realizations

In the previous sections, we have shown four different realization for the ELM. Each realization has its strength in terms of efficiency of memory usage and computational complexity of address the memory. In this section we make some general remarks about the different realizations and summarize the strengths and weaknesses of the four realizations.

A. General Remarks

i. Linearization Shapes

In the realizations, the Linearization shape of the OPD determines the complexity of the *getReadAddress* implementation. We indicated that when rectangles are enforced, simple polynomials result. There are application domains where the rectangular shape is the natural Linearization shape, for example, in imaging. In those cases, the Linear realization doesn't have the disadvantage of memory wastes and the need for a $recover$ function. On the other hand, we found more complex Linearization shapes in advanced signal processing algorithms. In algorithms like QR or SVD, triangular shapes are typical leading to complex pseudo polynomials.

ii. Parameterized versus Static realizations

We solved the Linearization under the assumption that we want to keep the problem parameterized in the original parameters of the loop-bounds of the parameterized NLPs. If, however, we need to provide a realization for specific values of these parameters, we can come up with a much simpler realizations of the Controller. The Segment realization already shows that. In general, if we can evaluated the *getReadAddress* a priori, the Controller becomes a simple look-up table.

iii. RAM versus CAM

RAM is the most commonly used form of memory. It is simple, cheap and widely available on todays FPGAs. But more and more, CAMs are also becoming available. Now a days, there are FPGA platforms available on the market that supports CAM blocks with high speed search time [15].

iv. Dense Polytopes

In the examples shown so far, we assumed that all nodes in the OPD and IPD can be enclosed by a Linearization shape. There are cases, however, in which we can find the exact shape, but still not all points are part of the enclosure. We refer to these points are *holes* and they are introduced when a for-loop is used with a stride other than one or when linear expressions are used that contain operators like mod, div, floor, ceil, max, or min. The holes affect the generality of the realizations presented in this chapter. Not all of the discussed realizations can handle holes. For example, if a Linearization shape encloses holes, these holes get also ranked, thereby disturbing the consecutive order.

v. Recover Function

In three of the presented realizations, the function *getWriteAddress* is based on the *recover* function. For each token read from the FIFO, this function recovers the iteration at which this token was produced inside the IPD. Basically such function duplicates the control from the IPD as a finite state machine, which can be computationally expensive.

Instead of using the *recover* function at the Consumer, another approach would be to tag the tokens produced at the OPD with additional information. In this way, the Controller and memory have the same function as the *matching unit* found in classical Dataflow architectures [16]. The problem in these matching units was to find a lower bound on its memory, such that a program wouldn't deadlock. We have shown in equation 7 that we can determine a lower bound on the memory such that no dead-lock occurs given the class of parameterized NLPs.

vi. Practical limitations

The Pseudo-Polynomial realization depends very much on the ability to calculate the $rank$ functions. We rely on the Polylib library [17], to compute the rank function. Although this library has proven to be quite stable and useful, this implementation of the Ehrhart theory is not always able to compute the $rank$ function. By selecting the Linear, the Segment, or the CAM realization, we are always able to come up with a representation of a KPN.

B. Summary

We have presented four different realization for the ELM. In Figure 12, we compare these realizations for the Producer/Consumer given in Figure 6.

Linearization Model	Memory size	N=8	Computational Complexity	recover	Generality
Pseudo	$N^2/2 + 1/2 \cdot N - 6$	30	$C1$	No	No
Linear	$N^2 - N - 6$	50	$C2 \ll C1$	Yes	Yes
Segment	$N^2/2 + 3/2 \cdot N - 4$	40	$C3 \sim C2$	Yes	Yes
CAM	$max(read - rank)$	10	$C4 \sim C2$	Yes	Yes

Figure 12. Comparision of the ELM realizations

The table shows the memory requirements in a symbolic way for parameter N and when $N = 8$, the computational complexity of addressing the memory (as done by function *getReadAddress*), whether a *recover* function is needed, and finally the generality of the approach. We can see that the Segment realization uses more memory than the PseudoPolynomial realization because the segment part consumes some memory. The advantage of the Segment realization is that the Controller can fill the memory in the same order tokens arrive. The Segment realization uses less memory than the Linear realization. If you look to the computational complexity of addressing in the Segment or Pseudo-Polynomial cases, you can see that the complexity is less for the Segment realization although in both cases a pseudo polynomial is evaluated. Finally, we observe that the CAM realization uses the least amount of memory but may require a relatively complex addressing mechanism since the CAM realization requires the computation of unique key.

X. Conclusions

In this chapter, we have presented the Extended Linearization Model (ELM). This model was introduced to solve out-of-order communications of tokens between a Producer and a Consumer process. The ELM adds to a process some additional memory and a controller without violating the Kahn Process Network semantics; we still use a FIFO between a Producer and Consumer. The Controller uses the local memory to re-order the tokens in the order the Consumer expects the tokens. In the realization of the ELM, the implementation of the Controller is the difficult part.

To implement the Controller, we make a lot of use of the $rank$ function. This function assigns to an arbitrary iteration a unique rank number that indicates when it is produced. The $rank$ function is in general a pseudo-polynomial. We exploit the ability to derive such polynomial at compile time to find realizations for the ELM at compile time. In the realizations, we assumed that we only exchange tokens over a FIFO without any additional information. We did not assume tagging of tokens.

When realizing the ELM, we have seen that four different realizations exist. The first realization is the pseudo-polynomial realization. It uses exclusively pseudo-polynomials to solve the reordering case. The advantage is that we can solve the reordering in a parameterized way. Because in the most general case pseudo polynomial are involved, the implementation can be computationally complex. Also, the pseudo-polynomial can in practice not always be calculated. If we relax the Linearization shape to the smallest rectangular that encloses all iterations, we obtain a more simple implementation. However, this might be at the expense of inefficient memory usage. If we want to avoid complex addressing and efficient memory usage, the Segment realization is a good choice, although the solution is not longer parameterized. Finally, we showed that we can use a key instead of an address to retrieve the proper tokens. The calculation of this key is in general a simple polynomial which is easy to realize. Also, the CAM realization requires the least amount of memory to solve the reordering of tokens.

The KNPs derived by Compaan can be simulated using the YAPI framework [18] or using the PN-domain in Ptolemy II [19]. In both cases, we have to implement the presented realizations in software. We are currently able to implement in software, at compile time, the PseudoPolynomial, and CAM realization. For these realizations, we have shown that we can derive correct implementations. We verified this by running Compaan on a set of applications written as parameterized NLPs.

References

1. Kienhuis, B., Deprettere, E., Vissers, K., van der Wolf, P.: An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97), Zurich, Switzerland (1997) 338 – 349
2. Kienhuis, B., Deprettere, E., van der Wolf, P., Vissers, K. In: A Methodology to Design Programmable Embedded Systems. Volume 2268 of LNCS. Springer Verlag (2002) 18 – 37
3. Kienhuis, B., Rijpkema, E., Deprettere, E.F.: Compaan: Deriving process networks from matlab for embedded signal processing architectures. In: 8th International Workshop on Hardware/Software Codesign (CODES'2000), San Diego, USA (2000)
4. Wolf, W.: Computers as Components - Principles of Embedded Computing System Design. Morgan Kaufmann Publishers, Inc. (2001)
5. Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE **83** (1995) 773–799
6. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of the IFIP Congress 74, North-Holland Publishing Co. (1974)
7. Harriss, T., Walke, R., Kienhuis, B., Depettere, E.: Compilation from matlab to process networks realized in fpga. Design Automation of Embedded Systems **7** (2002)
8. Ben-Ari, M.: Principles of Concurrent Programming. Prentice Hall

(1982)

9. Turjan, A., Kienhuis, B., Deprettere, E.: A compile time based approach for solving out-of-order communication in kahn process networks. In: Proceedings of the IEEE 13th International Conference on Application-specific Systems, Architectures and Processors (ASAP'02), San Jose, California (2002)
10. Ehrhart, E.: Polynômes arithmétiques et Méthode des Polyédres en Combinatoire. International series of numerical mathematics vol. 35 edn. Birkhäuser Verlag, Basel (1977)
11. Clauss, P., Loechner, V.: Parametric analysis of polyhedral iteration spaces. Journal of VLSI Signal Processing **19** (1998) 179–194
12. Rijpkema, E.: Modeling Task Level Parallelism in Piece-wise Regular Programs. PhD thesis, University Leiden, LIACS, Leiden, The Netherlands (2002)
13. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986) ISBN 0-201-10088-6.
14. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, Inc. (1997)
15. Xilinx: Memory Application Notes for Virtex-II Devices. www.xilinx.com (2001) pag 505-506.
16. Veen, A.H.: Dataflow machine architecture. ACM Computing Surveys **18** (1986) 366–396
17. Clauss, P., Loechner, V.: Polylib. http://icps.u-strabg.fr/Polylib (2002)
18. de Kock, E., Essink, G., Smits, W., van der Wolf, P., Brunel, J., Kruijtzer, W., Lieverse, P., Vissers, K.: Yapi: Application modeling for signal processing systems. In: 37th Design Automation Conference, Los Angeles, CA (2000)
19. Davis II, J., Hylands, C., Kienhuis, B., Lee, E.A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Tsay, J., Vogel, B., Xiong, Y.: Ptolemy ii - heterogeneous concurrent modeling and design in java (2000)

A scalable platform for large scale array signal processing systems specification and prototyping

Sylvain Alliot
ASTRON, Netherlands Foundation for Research in Astronomy
Dwingeloo, The Netherlands
alliot@astron.nl

Ed Deprettere
LIACS, Leiden Institute of Advanced Computer Science
Leiden, The Netherlands
edd@liacs.nl

Abstract

In order to provide means to comparing functional and structural alternatives in an early stage in the design of large scale array signal processing systems, in particular phased arrays for radio astronomical observations, and to identify and isolate the functional and structural critical parts of the specification of the system, we have specified, designed and implemented a scalable platform that demonstrates the feasibility of a proposed specification and design exploration methodology. The platform can be used for experimentation by the end-users whose findings will be taken into consideration during the development and implementation of the ultimate exploration framework. The Thousand Element Array (THEA) *platform is a downscaled version of huge distributed radio telescope which will be specified, designed and built in the next decade. It performs beamforming, adaptive mitigation of radio frequency interference (RFI) and correlation for noise suppression. The THEA digital back-end is a composition of blocks that are themselves heterogeneous compositions of modular and programmable/configurable library elements. The blocks come in families and instances of them can be used for the specification and design of a large set of telescope configurations. The data interfaces in particular permit multiple block combinations for optimal data routing at system level. System composition is thus simplified and facilitates the system specification, exploration and design tasks of the designers, given the requirements and constraints provided by the end-users. A set of specific hardware boards has been implemented which are hardware instances of members of the families of blocks that were proposed after a careful analysis of a range of different radio telescope requirements and constraints for families of applications. The hardware blocks that have been selected for the THEA platform have been a compromise between performance and flexibility objectives to allow the end-users to tune their requirements and to validate the system. Moreover the key issues of a large scale system such as data management, processing and control can be demonstrated with the current platform.*

Introduction

Astronomers from all over the world are currently regularly meeting to discuss requirements, constraints and boundary conditions for a number of very large radio telescopes that have to be built in the coming twenty years or so. Examples of such telescopes are the *Atacama Large Millimeter Array* (ALMA), [1], the *Low Frequency Array* (LOFAR), [2], and the *Square Kilometer Array* (SKA), [3]. Besides being very high speed imagers, LOFAR and SKA will include hierarchical beamformers and adaptive filters in space, time and frequency. These telescopes will be composed of large clusters of distributed antenna elements that will be built on hierarchies of embedded systems with hard real-time and very high throughput constraints, and data intensive processing capabilities. However, astronomers do not provide system specifications. They only give their input-output relations, modes of operation, and a number of metrics that they can use to evaluate alternative system specifications and system designs. Moreover, the given requirements and constraints may be incomplete or open for modifications at the time the designers - system designers as well as software and hardware designers - take off for their parts of the complex task of building such large systems. Thus, this process is not a chain of consecutive actions but rather a action graph with many concurrent tasks and unavoidable dependencies between these. Such huge project can only be undertaken with confidence if the design of such system is a well structured, interactive and iterative trajectory that is based on a sound methodology. Indeed, what the astronomers as initiators as well as end-users of the system would like to have to their disposition is a *transparent exploration framework* that they themselves can use to answer their own *what if* questions. What if the cost is too high? What if we change the required resolution? What if we add this or that mode of operation? Developing and implementing such an exploration framework is itself a major effort that is to be undertaken well before the telescopes are in place. Work in this direction is currently in progress, and although this paper is not intended to elaborate on the underlying methodology it makes sense to briefly sketch it here because the THEA platform which is the main subject of the paper, has been both a test case and a driver for the methodology. Thus the building of a huge distributed radio telescope goes in three equally important and partly paralleled phases: 1) the exploration of the specification space that emerges from the system requirement parameters and constraints, 2) the exploration of the design space that is defined by on the one hand, the model in which the specification is casted and, on the other hand, the library blocks based implementation platform model into which the specification is to be mapped, and 3) the implementation and realization of the final system itself. The first two phases are iterative and interactive processes that are supported by fast simulation and metrics based performance analysis tools and methods to accelerate the exploration trajectories. This can only be achieved if the explorations are conducted at high enough levels of abstraction without delving too deeply in the details. It has been demonstrated by several researchers that exploring at higher levels of abstraction has greater impact on performance improvements and cost reductions than can be obtained at the lower levels of abstraction. However, performance and cost measures at the highest level of abstraction are partly imported from the lower levels, and for these to be of sufficient confidence, critical parts have to be taken down to lower levels of abstraction where some sort of fine calibration can be done and from where performance and cost numbers can be safely injected in the higher levels of abstraction. The THEA platform has been the first attempt to specify a scalable telescope prototype system for which the SKA telescope was taken as the large scale telescope that would then have to be shown to admit a specification that is a scaled version of that prototype. It has been shown by one of the authors that this could indeed be done. Two remarks are in place here. Firstly, not much exploration has been done to specifying the THEA platform because no exploration framework was available yet. Only some sort of *golden point* specification was aimed at. Secondly, the actual implementation of the THEA specification as a prototype has demonstrated that without going through the design space exploration phase, the direct implementation of the output of the specification exploration phase yields a system that satisfies the requirements of the end-users. With direct implementation we mean an implementation that is composed of those off the shelf components that were used in the calibration steps of the specification. The rest of the paper is organized as follows. In Section 1, we present the basic trajectory that takes requirements to specifications. In section 2, we introduce the formation of basic blocks into which processing tasks can be mapped for calibration. Finally, in Section 3, we come to the way in which blocks are interconnected to reach the system level specification. Recall that all this is in the context of the design of the THEA platform. The fore-mentioned design for implementation trajectory, in which predicted technology advances are taken into account, has not been considered. Instead, the specification was mapped directly into the blocks that have been specified in the specification phase in terms of heterogeneous architectures in which the components were off the shelf programmable and configurable devices.

1 Specification trajectory

Deriving a system specification from user requirements is essentially a structured iterative process. It is structured in the sense that it is expressed in terms of available models and methods at a number of abstraction levels that have well defined intra level and inter level interfaces. Reuse of software and hardware entities is usually a condition and these entities must themselves be so specified that they can be easily imported in the system specification with standardized interfaces and easy to integrate internal measures of performance and cost. It is iterative because decisions made at each and every level of abstraction are dependent on higher level decisions and have impact on lower level decisions. It is a typical meet in the middle process in which the number of abstraction levels that is visited depends on the degree of confidentiality that the end users require the specification to be presented to them by the designers. In this section, we take a quick look at this process, and we present the THEA platform that is used to run through that process.

1.1 The global picture

As shown in Figure 1, deriving a system specification from the given requirements is a process that runs iteratively through several levels of abstraction.

First, the user defined requirements that come as input-output requirements, constraints and boundary condition, are translated into a high level specification that is a functional and structural decomposition of these requirements: a network of communicating and interacting behaviors. That decomposition is not unique and, therefore, some exploration and metrics based analysis is in place. Issues of concern are sensitivity, concurrency, orthogonality, and computational complexity and variability of the individual behaviors in the network. Measures such as latencies of tasks and throughput of signals are still not available at this level, they instead are derived constraints.

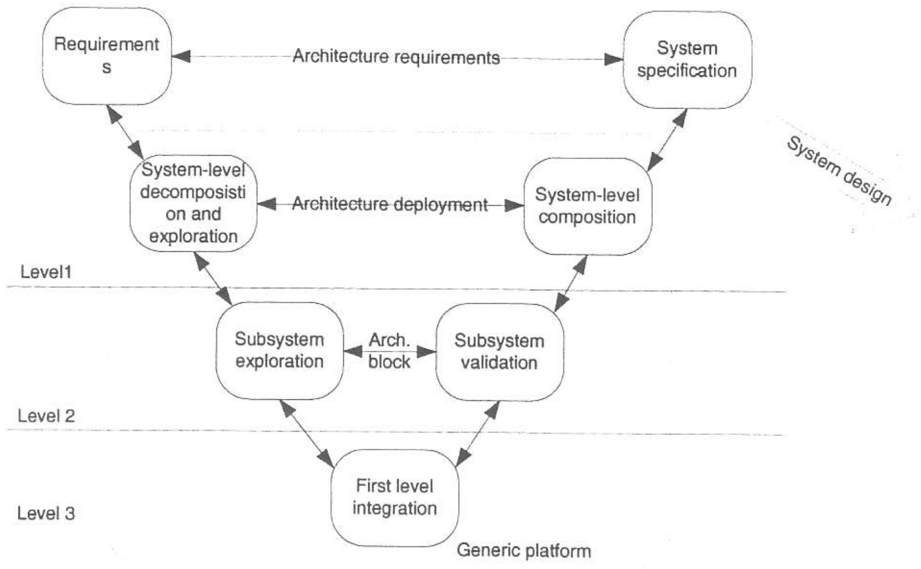


Figure 1: A design trajectory to extract specifications from requirements and the different levels of abstraction for the application to architecture mapping, application decomposition and composition

Nevertheless, there is a rough indication of computational latencies based on counts of atomic operations in the executable specifications of the behaviors. Next, critical subsystems are identified and taken one level down the abstraction hierarchy for further decomposition and exploration. This provides refined indicators of performance and cost. If necessary, this process of identifying critical parts, refinement and further exploration can be repeated until a level is reached where a final calibration can be performed. This calibration consists of mapping of (parts of) behaviors at this level of detail into state-of-the-art components and obtaining sufficiently accurate numbers expressing performance and cost. These numbers are then exported back into the first higher level of abstraction where the performance and cost measures are expressed as performance and cost numbers of blocks that are heterogeneous compositions of the lower level components. The stage in which blocks are defined and specified is denoted *level 3* in Figure 1. From there on, all higher levels are specified, recursively, in terms of compositions of blocks. The blocks so defined and specified are members of families of blocks that are made available in a library. This library, together with block connection rules and methods to obtain performance and cost measures of the block compositions from the performance and cost measures of the block themselves constitute a platform of which various instances can be selected and analyzed.

1.2 The platform

During a feasibility study, the requirements are changing due to feed back to and interaction among the users and experts. This is not only the case in the specification phase, it is likewise so in the design phase because of emerging new technology or even because of unexpected modifications in constraints and boundary conditions which may emerge from the users side or from the designers side. With a platform, including a transparent exploration framework, the users as well as the designers are given options to reconfigure the system specifications fast and to adapt to such changes. The modifications may enter at any level of abstraction and may ripple down and up the hierarchy; in all cases, the end-users must get answers to their *what if questions* in a reasonably short time and with a reasonable degree of confidence. What does the platform consist of? It is a framework that contains libraries of entity specifications, rules by which entities can be imported in models, means to evaluate performances and costs of the models, means to explore alternatives, and feedbacks to support a fast exploration process that can yield feasible solutions to the specification (modification) or the design (modification) problems. For the THEA platform, one library contained executable specifications (written in the Matlab algorithm development environment) of signal processing tasks, such as filters in space, time and frequency, Fourier transforms, adaptive RFI suppression algorithm and correlation algorithms. Another library contained off the shelf components, such as DSPs and FPGAs. Yet another library contained families of blocks that are heterogeneous architectures built on the off the shelf components. Also available were means to map (parts of) signal processing algorithms into the components to extract performance and cost numbers at this level of abstraction. The interfaces of the blocks were so defined that interconnection of blocks yield correct compositions by construction with throughputs as required. Scalability is also guaranteed at this level. For the THEA platform case, blocks were not readily available and have had to be designed and specified while developing the platform and served to evaluate the various THEA digital back-end applications, such as the multi-direction beamformer [4] and the spectrometer [5]. These blocks have also been shown to be consistent for the specification of the ALMA correlator [6], and the LOFAR and SKA stations [7]. The blocks have been specified and designed with a good compromise between programmability and efficiency in mind. They are crucial entities in light of the required scalability of the platform and, therefore, we take a closer look at them in the next section.

2 The THEA building blocks

The building blocks that have been specified for the THEA platform are intermediate entities that are suitable for the specification of larger systems. To be more specific, assume that the specification of a subsystem of a large system is given in terms of three models: a behavioral model, for example a Kahn Process Network [8], an organizational model, for example a set of building blocks embedded in an interconnection network, and a mapping model that relates the two other models. Then, after selecting values for the parameters in the three models, an instance of the platform is created and the evaluation of that instance provides a complete specification instance for the subsystem. The resulting platform instance is not necessarily an implementation architecture - though it can be physically realized (as it has been), it is in general, a model of specification in its parts of which the necessary metrics can be evaluated, and that, at the same time is within the constraints and boundary conditions imposed by the designers themselves: what can be done, and also what cannot, is well defined and well structured. Once the subsystem is specified that way, the larger (sub) system can be easily specified as well because the platform is scalable: Kahn Process Networks are compositional and so are the other two models and, hence, the platform. In the next several subsections we zoom in on the block structure, the internals of a block, and the notion of families of blocks.

2.1 The block structure

In Figure 2 a typical building block is shown as a hierarchy of elements in a class diagram. This object oriented structure makes the evaluation of performance and cost numbers easy because those are inherited from the elements forming the block.

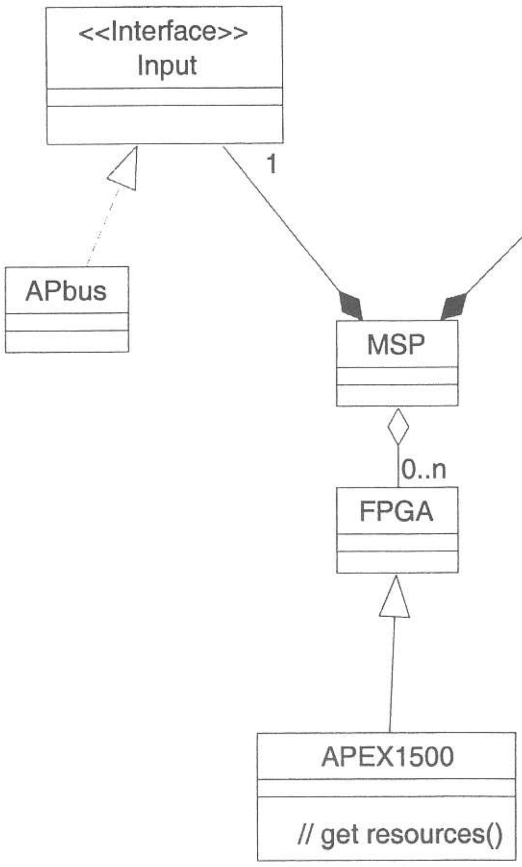


Figure 2: A building block specified as a UML class diagram

As an example here, the class MSP is constructed with the aggregation of interfaces and FPGAs classes. A specific interfacing object called APBus that represents a particular physical bus has been attached to the interfaces and n FPGA objects of type APEX1500 are created. The methods of the MSP class can access the objects methods that are linked to mapping information of a process network, or a combination of them, on a architecture block. The component's static properties are also easily inherited. As an extension, the combination of these within a model at a block level is accessible for higher system architecture constructions out of blocks such as MSP.

Another view is proposed in Fig. 4 to represent the blocks and the elements of a block within a particular topology. This determines the composition of a block and it introduces the constraints of the internal data distribution. The properties of the block related to the topology are taken care in the performance evaluation. As defined later on, when the topology of the blocks are similar, the block belong to families sharing the same rules for performance evaluation.

Because any platform is application specific, the building blocks here have also been defined and specified with the radio astronomy applications in mind. These applications are characterized by huge data bandwidths, very high data throughputs, and processing requirements that are increasing in volume, quality, and flexibility. Thus, the digital part of the platform must not only sustain high throughputs and data intensive computations, but also high level real time control for interaction with a remote front-end and a distributed computing facility. The dominating requirements, however, are the quality, speed, and accuracy of the computations and the interconnections. With respect to the building blocks themselves, these requirements are partly inherited from their constituent components and partly embedded in the way these components are interconnected. As a result, the building block components had to be carefully selected as well. Recall that the building blocks as well as their constituent components are parts of the organizational model in the three-model platform. Processes and channels in the behavioral model are mapped, using the mapping model, onto these building block components that have timing behaviors, performances and costs that are known from their specifications in the component library. These specification may be given in terms of models and methods, that may or may not correspond to physical devices. In the case of the THEA platform, the components have been the entities at the lowest level of abstraction in the organizational model and have been modelled after physical devices. The next subsection deals with the rationales after the selection of these components.

2.2 Selection of components

Given that the application domain is advanced radio astronomy, the execution of several signal processing algorithms in real-time and at a sustained high throughput, the components that have been selected are Digital Signal Processors (DSP), Memories, Interfaces and Interconnection Networks. Standard CPUs have also been selected for less demanding control processing. Because CPUs and DSPs have fixed architectures and standard communication busses, and Memories are usually large lumped memory banks, those components do not form a complete set as they are excluding mixtures of processing and interconnections in a distributed way could be efficiently mapped into them. Therefore, FPGAs have also been selected as component because they have no fixed architectures, and fast and efficient communication networks in which high throughput processing units and distributed memory can be embedded and easily configured. Of course, the building blocks as such are not specified after the components have been selected. Recall that components are calibration elements in the specification phase. Thus some algorithm and mapping exploration must be carried out to validate the calibration procedures and to provide performance and cost numbers for alternatives on that level of abstraction. Here is an example. An adaptive RFI mitigating beamformer consists of two parts: A space filter, which in essence is an inner product of two complex valued vectors - say of 16 entries each, viz., a signal vector output by the antenna array, and a weight vector that is varying in time to track and null the RFI in the beamformer, and an adaptive-weight estimation algorithm that figures out what the optimum weight vector should be for effective RFI nulling. On passing, it is instructive to note that human made radio frequency signals may peak up to 70 dB above the ambient noise level, whereas the signals of interest that come from fainted stars and galaxies may delve down to 70 dB under that noise level. Now, a measure of complexity of the complex valued inner product is easy to obtain from the specification of that function in the behaviorally model of the platform: it is 16 complex multiplications and 15 complex additions. The calibration of that function in the organizational model of the platform is easy in case the function is to be mapped into a DSP: The specification of any DSP includes the performance and cost for that function. However, for THEA as for the other systems of concern here, no DSP can achieve the fast pace at which consecutive inner product evaluations have to be executed. Therefore, a calibration in an FPGA may be a better choice. Indeed, it is relatively easy to map the inner product as a systolic array into an FPGA. In this case, it is even not necessary to do so, because the specification of any FPGA will provide cost and performance of a single complex multiply-add pair, and a simple extrapolation can yield sufficiently accurate performance and cost measures, not only for the space filter itself, but also for the achievable throughput. This is what has been done for the development of the THEA platform. No further alternatives have been explored. See e.g., [9] for an exploration case on executable specifications of the inner product. The situation is dramatically different and more complicated for the adaptive weight estimation algorithm. Indeed, there are many adaptive weight estimation algorithms all deferring in terms of accuracy, complexity, latency and throughput. Figure 3 shows the minimum execution time for three different weight estimation algorithms when using a DSP for calibration (or implementation). If higher accuracy or faster update rates are required, other algorithms have to be explored and other calibration or implementation (models of) devices have to be conceived.

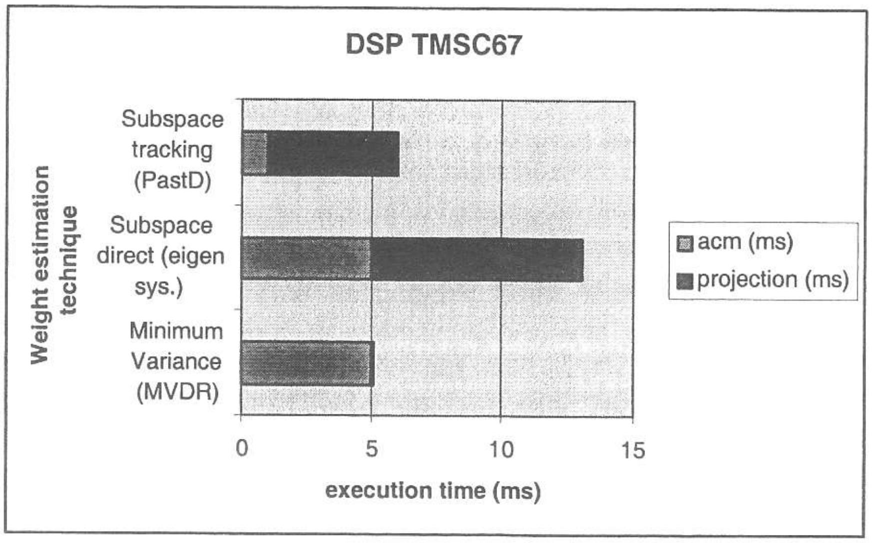


Figure 3: Example of exploration: the adaptive weight estimation in the THEA spatial RFI nulling using a floating point DSP for calibration/prototyping for three different nulling algorithms

Although the number of components that have been selected for the THEA platform specification (and prototyping) is reasonably small, there are still many ways in which these components can be interconnected to be promoted to building blocks. It would be very appealing if one could do with only one building block but that is too optimistic. Indeed, executable specifications of signal processing algorithms that can be mapped on components are commonly tasks or processes that occur in process network specifications that are decompositions of higher level processes. It is typically these higher level process decompositions that are mapped on building blocks. However, different higher level processes have unequal process network decompositions and widely varying constraints that are imposed on them. At first glance, one might expect that a large number of building blocks would have to be available in the block library. It turns out that the definition and specification of only a moderate number of building blocks is sufficient. More precisely, only a handful of building blocks or, more precisely, of building block families are needed. We shall take a closer look at these families in the next subsection.

2.3 Block families

Roughly speaking, a block family relates to a platform much as a building block relates to a platform instance. That is to say, a block family is a parameterized heterogeneous collection of components into which certain specific behaviors can be mapped effectively and efficiently and others not. The block's structure induces data exchanges properties from and to elements in a specific way. Such properties are common within a family. Therefore the families are the abstraction level for a block composition. A member of the block family distinguishes itself from another member in that it performs better or worse and at a lower or higher cost than other members. Table 1 lists the families that have been identified during the development of the THEA platform. The symbolic representation of members of these families is shown in Figure 4.

The block definition results of a separation and aggregation of architectures and application likely to be most efficiently hosted by a generic architecture design as presented in Tab. 1. The techniques for efficient processing on the generic elements impose constrains that are element and block combination specific. Defining a generic block is a trade-off to satisfy the main applications requirements. Such requirements are data exchange related : memory and interface location and access and processing power related : distribution, pipelining dependencies. Therefore to enlarge their usage, the platforms have been built with a large number of connections and the fastest interfaces available at a board level. Thanks to the latest chip density improvements the cross connections could be limited mainly to a single integrated board. The re-scaling of the system according to new technology shifted most of the data routing complexity in individual modules, thus drastically improving the system performance and robustness.

We take a closer look at them in the next six subsections.

2.4 Detailed description

2.4.1 Matrix shuffling/processing

The block called MSP is emphasized in Fig. 2. It can be used as a basic shuffling device crossing a great number of lines. Filters, cross and auto-correlators or coherent adders are examples of applications with vectors or matrices as input formats. Their refinement into process networks can be matched to a matrix network with elementary processing nodes. The regularity of the application and the chip internal structure of a FPGA are therefore very similar. At board level, a multi-FPGA system with a matrix network topology is the extension of the optimum system

Table 1: Families of building blocks from which building block instances have been selected for the specification of the `THEA` platform.

Block family	Target processes
Matrix shuffling / processing	Central processing Cross-correlation Adaptive cancelling / nulling Beamforming
Fast pipeline processing	Distributed processing Data flow online filtering Spectometer
Fast control processing	Complex memory circulation Array decomposition and processing Space / time / frequency analysis Application control
High speed link	Mass data transfer Cluster connection Mass routing RT control lines connection Remote data transfer Remote RT control
Selection	Massive Cache-storage Data Multiplexing / routing
IDC	Band mixing Band separation Analog / Digital conversion
Main acquisition and control	Monitoring Network support Hardware support

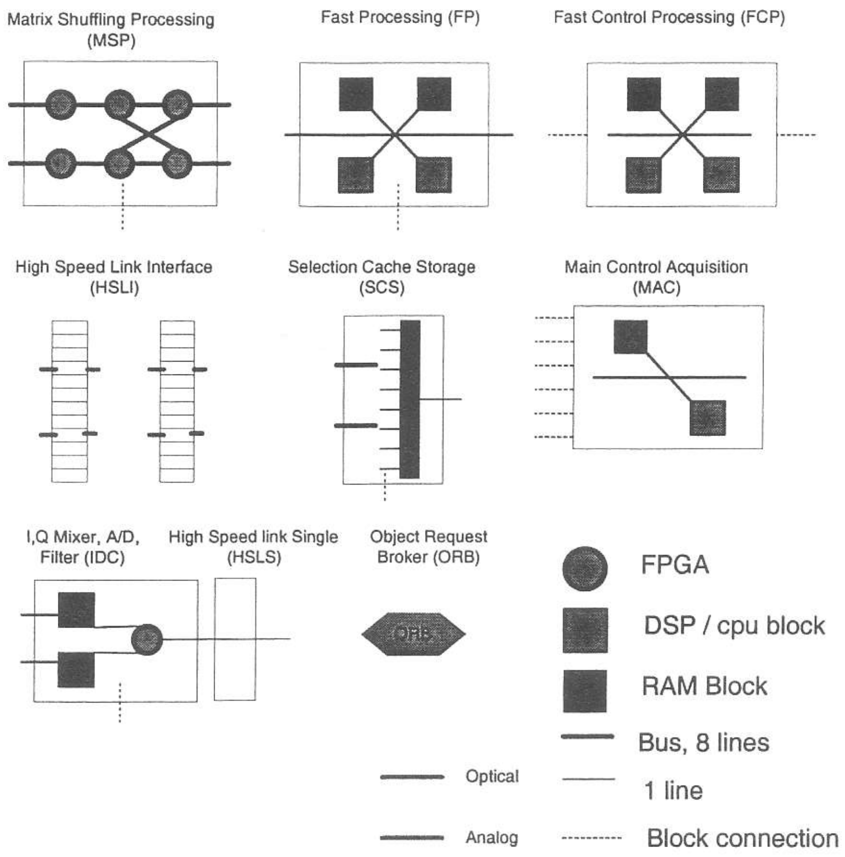

Matrix Shuffling Processing (MSP)
Fast Processing (FP)
Fast Control Processing (FCP)
High Speed Link Interface (HSLI)
Selection Cache Storage (SCS)
Main Control Acquisition (MAC)
I,Q Mixer, A/D, Filter (IDC)
High Speed link Single (HSLS)
Object Request Broker (ORB)
ORB
FPGA
DSP / cpu block
RAM Block
Bus, 8 lines
1 line
Block connection
Optical
Analog

Figure 4: Platform scalable blocks

on chip implementation. The mesh topology [10] was used for very large scale implementation of a single application over a board or multiple boards such as telescope correlators [11]. A variant with a radix topology is proposed for the current platform enhancing the cross node flow for multi-beaming operations [4]. Matrix shuffling within a mesh is coupled with large processing capabilities at the knots. The integration and the large number of pins in a FPGA, make this device very attractive for the data crossing on chip. Most of the elementary processing on the data flow can be efficiently distributed on the board limiting the intermediate data storage.

Table 2: Matrix shuffling processing block

Current features	Standard block
6 FPGA, APEX 400-1500 (Altera)	Mesh processing topology
I/O : 384 lines at 40 MHz, 16Gbit/s	Large pipelined I/O rates
Ext. bus : 32bit 10MHz [12]	Block I/O transfer (control/acquisition)

2.4.2 Fast pipeline processing

If the array is made large, the internal and external memory bandwidth in a FPGA is decreasing and deteriorates the system's performances by spanning the memory bus over the array, this is demonstrated in [13] [14] and can not be easily solved with hand optimized memory skews. When matrix shuffling is not a main task for instance for online filtering, fast real time digital signal processing is required with immediate access to large RAM blocks. For the THEA spectrometer application, the memory constrains were higher than the largest FPGA internal memory and DSP structures were selected instead. The fast processing is inserted in the data pipeline processing blocks. The sustained telescope high data rates are reached by duplicating the processing units on blocks of data with fluid access to dual temporary memory and interfaces. The architecture with interfaces, memory and processing elements routed with a cross bar element is easily configurable and can host a large variety of applications. The level of computation is simply raised by duplicating the number of processors if required. Such systems on chip are proposed in the next generation of FPGA [15]. After prototyping of the acquisition routine as well as the processing on a DSP chip for the spectrometer application, a block with sufficient processing and memory width was commercially available for THEA at the scale of a board called PMP8 [16] see performance in Tab. 3. The fast processing unit can be pipelined or connected to a common bus for additional processing power.

2.4.3 Fast control processing

Control processing stands for advanced processing with complicated arithmetic or control functions. This block is directly connected to the devices handling the data flow and consequently is real time specific. Nevertheless, the control processing is not directly receiving the raw data. For THEA, the control processing constrains were high and had to be implemented in a guaranteed real time environment like a fast DSP tied loop to the row data stream. Eventually, The processing-control could also be shared by the PC CPU with lower response constrains. The fast

Table 3: Fast pipeline processing

Current features	Standard block
8 DSP TMSC6201	Dedicated for distributed processing
500&250 MB/s burst interface	Fast sustained I/O rates
2 independent 128 MB SDRAM	Large memory resources/high bandwidth
1 DSP TMSC6201 / 6 points cross-bar	Multiple cross-bar/controller/sequencer
Ext. bus : PCI 32bit 33MHz	Block I/O transfer (control/acquisition)

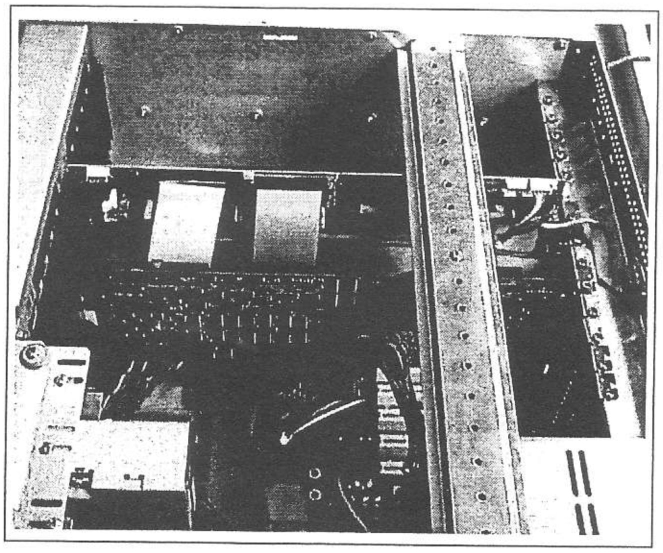


Figure 5: Industrial PC host for 16Gflops fast processing platform and data acquisition

PCI connection and fast dedicated bus to a DSP for external communication was commercially available. The Daytona board from Spectrum is a real time control board for relatively fast and complex response see features in Tab. 4.

2.4.4 Acquisition / main control

The system infrastructure is often an important cost driver in back-end designs. It was chosen to simplify the system composition by separating processing nodes with independent control and acquisition capabilities. The personal computers are in that respect capable of hosting multiple embedded systems on a PCI bus with control and power supply. VME and compact PCI were not chosen because of their higher cost for equivalent or lower acquisition performance. The hardware support of THEA is an industrial PC for fast access to the control and processing boards for data acquisition via the PCI bus. The PC is connected to an Object Request Broker via an ethernet network for a central control with a client server application.

Table 4: Fast control processing

Current features	Standard block
2 DSP TMSC6701 (float)	Accurate / fast complex processing
4 points Cross bar	Dedicated for data / memory circulation
2 independent 64 MB SDRAM	Large memory resources / high bandwidth
2 independent 1 MB SRAM	Fast memory addressing / high bandwidth
Ext. bus : PCI 32bit 33MHz	Block I/O transfer (control/acquisition)

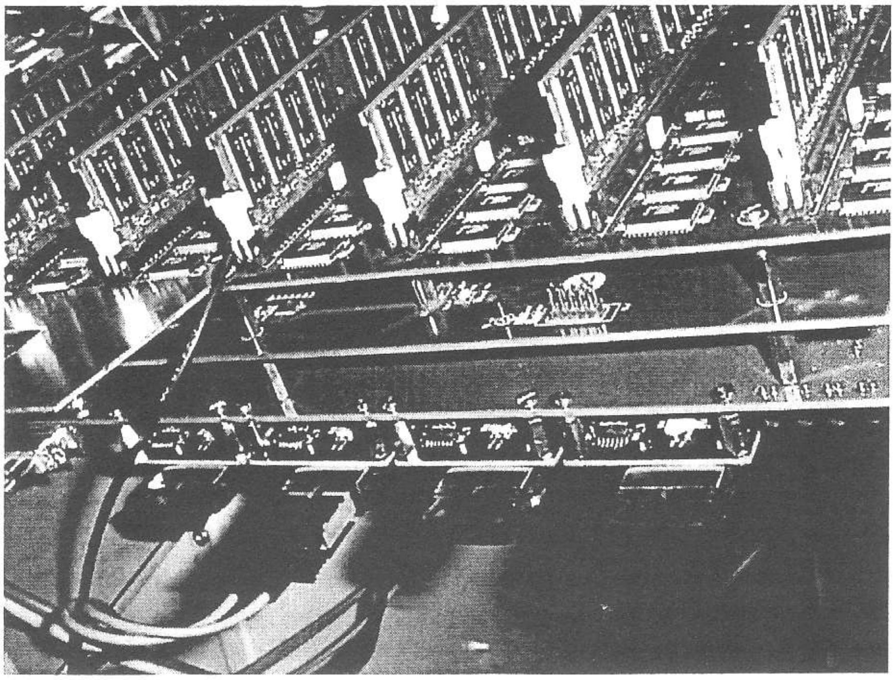


Figure 6: Combination of 3 blocks. SCS, MSP, HSL

Table 5: Acquisition main control

Current features	Standard block
Pentium III	Support: hardware/communication/scheduling
100 Mbit/s ethernet connection	Standard network interface
Ext. bus : 8 x PCI 32bit 33MHz	Multi-Blocks I/O transfer(control/acquisition)

2.4.5 Selection, cache storage

The selection is a new block compared to traditional designs, it gives the flexibility to map different applications' specific data reduction schemes. This module can act as a cache or re-circulation memory for the entire system at the antenna sampling frequency full resolution. The data acquired can then be transmitted at a lower rate to a fast processing platform.

Table 6: Selection cache storage

Current features	Standard block
8 * 64 bit SDRAM modules	Cache storage / large I/O rate
In: 384 lines at 40 MHz, 16Gbit/s	Time / channels multiplexer
Out : 160 MB/s synchronous interface	Independent scheduled controllers
Ext.bus:PCI 32bit	Multi-Blocks I/O transfer

2.4.6 High speed link

Given the high distribution rates, specific high-speed links have been developed for remote connection. The high speed modules are also a solution for optimum modularity. Each of the blocks described previously are modular with zero delay, zero electric disturbance point to point links that can easily connect multiple modules or multiple instruments. Clock and synchronization pulses are concurrently sent on an additional fiber, the system can consequently be distributed physically far apart along with every data and synchronization lines.

Table 7: High speed link

Current features	Standard block
Giga bit ethernet Tx, Rx	Serial/parallel interface
out : 10 clock + synchro pulse	Multiplexers
out : 384 lines / 40 MHz	Independent I/O channels
in : 10 x 1 Gbit/s	Domain interface (Ex: opt./ dig.)

3 A scalable back-end

3.1 Block connection

A system is scalable if its elementary blocks can be extended. For the generic platform the raise in processing capacity is given by a connection to another identical block. Consequently, the primary processing blocks of the THEA back-end are input/output symmetrical, as shown in Fig. 8 for cascading of processing capacity without obstructing the data flow. In some cases, the high speed data link interfaces can also be used to extend a matrix connection. The industrial PC, compact, widely spread and easy to connect via ethernet boards on a dedicated network is

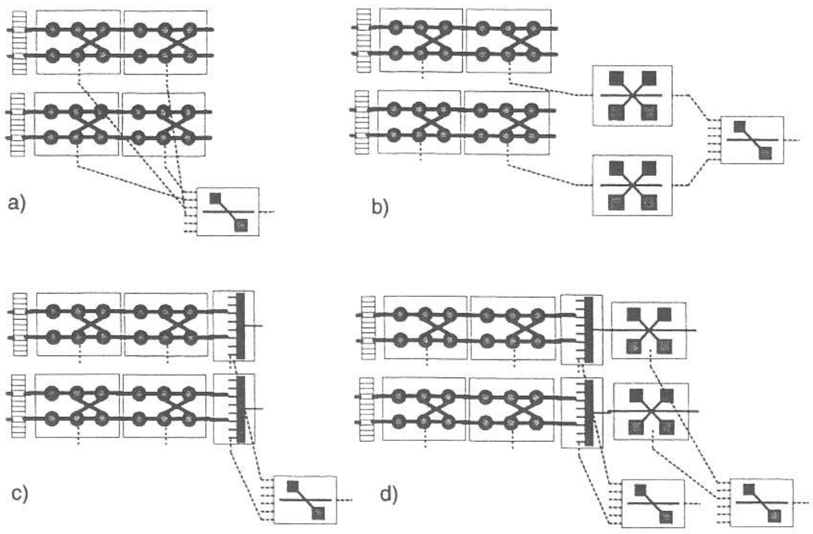


Figure 7: Different acquisition architectures

the host for the more dedicated hardware. Indeed, the PCI bus can distribute the power and the non time critical data.

3.2 Data routing

The control is strongly pyramidal and the data flow is linear. Actions are pipelined synchronously and some mechanize data crossing. Therefore, one of the main challenge in setting a generic platform was the design of elementary and complex processing with very large number of lines 392 data bits at 40 MHz for data crossing. For that reason the balance between the data flow and the processing is of prime importance. The approach here was to build modules based on the fastest interfaces and the maximum data flow on a board for a given application. The phased array beamformers and interferometers back-ends in radio astronomy in general are measuring cross products of elementary channels at very high bandwidth. The new generation of FPGAs such as the APEX 20K from Altera and VIRTEX from Xilinx, permits extremely flexible data routing within a single chip. Unfortunately, telescopes constrains are following Moore's laws and the systems are still spanned over several chips, boards and clusters of boards. The THEA beamformer is an application example overtaking the capacity of a single chip but a combination of 6 chips on a board (see [17]) allows optimal routing and maximum integration.

3.3 An object oriented control approach

The generic platform is a physically distributed embedded system that can be connected to a hierarchical scalable control system via network interfaces illustrated in Fig. 9. This approach for the control of THEA called TECH, described in [18], is implemented by a peer-to-peer communication through distributed Corba objects [19] as shown in the deployment diagram in Fig. 9. It offers transparent access to data and methods from all processes. The back-end blocks or objects from a software point of view are connected to TECH. Each block is embedded in a device driven

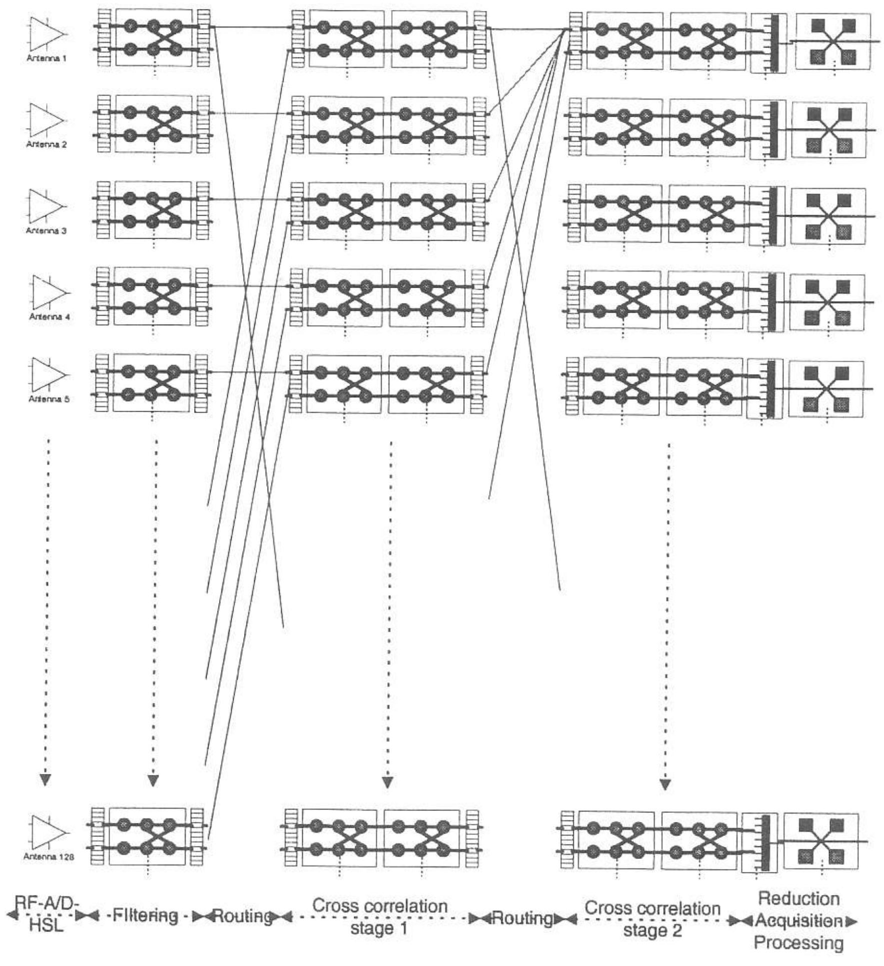


Figure 8: Scaling of the THEA blocks for a correlator

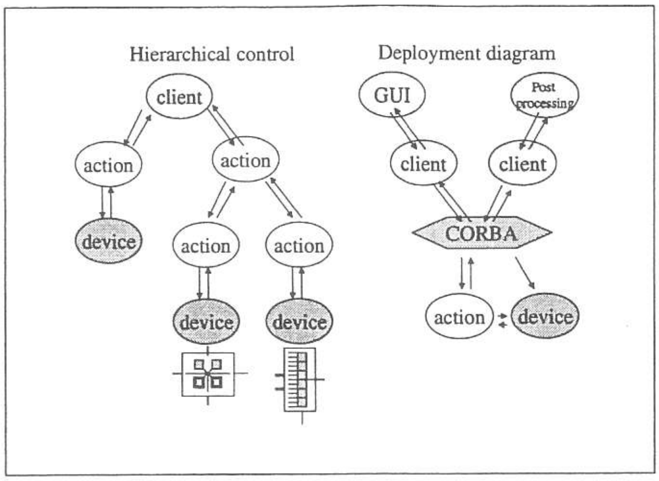


Figure 9: On this figure the tree of the object oriented software control is shown as well as the control deployment via CORBA for concurrent client applications

by actions. Consequently, the platform can be extended and upgraded from or within a pyramidal architecture. This structure is adequate for team work on the project but also for maintenance. As a base for a more generic back-end, independent drivers can be re-used for a different application.

Conclusion

We have presented the `THEA` platform which is a case study in ongoing investigations dealing with methodologies to designing specifications from requirements and implementations of the specifications. The implementation of such a methodology should provide a transparent framework for fast exploration and getting user satisfying answers to the user's `what if questions`. The development and implementation of the `THEA` platform has been a first and successful trial to structure the whole specification and prototyping project. The configurability and programmability of the platform offer exploration and validation facilities, not only for the relatively small scale Thousand Element Array, but for the distributed signal processing functions in all currently investigated radio telescopes. The platform is currently used as a versatile back-end for a telescope connected to the necessary control of the instrument. In the context of the the `ALMA`, `LOFAR` and `SKA` feasibility studies, different applications and technical solutions were proposed by different groups. A comparison of these proposals based on the `THEA` methodology and platform helped at an early stage of the project to establish specifications. Requirements, processing techniques as well as technology are likely to change within the development and first integration steps of the different large telescopes, yet the impact of the modifications can be evaluated and anticipated using such scalable platform of which the `THEA` platform is exemplary. What remains to be done is to improve the exploration part of the specification and design methodology. Work in this direction has to some extent been done [9] and the expertise gained with the `THEA` platform has been of great help to specify what exploration methods and tools are worth focusing on in order to make a platform based design of next generation telescopes really useful for those who provide the requirements and will be the users of these telescopes.

Acknowledgments

This study was performed in part of the MASSIVE project, funded by Progress and the THEA project at ASTRON funded by NWO. We want to thank especially the team that worked on the back-end of THEA : Dion Kant, Gijs Schoonderbeek, Harm-Jan Pepping, Ronald de Wild, Leon Hiemstra, Andre Gunst, kield van der Schaaf and others.

References

[1] R,S, Booth, "ALMA, The Atacama Large Milimeter Array," *Perspectives on Radio Astronomy: Technologies for Large Antenna Arrays*, pp. 17–22, April 1999.

[2] J. D. Bregman, "Concept design for a low frequency array," *Astronomical Telescopes and Instrumentation 2000 - Radio Telescopes, SPIE Conference 4015*, March 2000.

[3] Arnold van Ardenne, "Active adaptive antennas for Radio Astronomy; results for the RD program toward the Square Hilometre Array," *Astronomical Telescopes and Instrumentation 2000 - Radio Telescopes, SPIE Conference 4015*, April 1999.

[4] G. Hampson, R. de Wild, and A. Smolders, "Efficient Multi-Beaming for the Next Generation of Radio Telescopes," *Perspectives on Radio Astronomy: Technologies for Large Antenna Arrays*, pp. 265–276, April 12-14 1999.

[5] S. Alliot, "THEA spectrometer," *ASTRON internal repport*, May 2000.

[6] S. Alliot, "A System Architecture Exploration for a Correlator Based on Generic Blocks," *ALMA Report unpublished*, September 2001. http://www.astron.nl/~alliot.

[7] S. Alliot, "Scalability and extension of the THEA blocks towards very large phased arry telescopes," *MASSIVE Report unpublished*, November 2001. http://www.astron.nl/~alliot.

[8] G. Khan, "The sementics of a simple language for parallel programming," *Proc. IFIP congress 74*, Jully 1974.

[9] Laurentiu Nicolae, Sylvain Alliot, Ed Deprettere, "Distributed Radio Telescope as Massively Parallel Computing System: A case Study," *Proceedings MPCS, The Fourth Iinternational Conference on Massively Parallel Computing Systems*, April 2002.

[10] S. Hauck, "The future of reconfigurable systems," 1998.

[11] A. Bos, "A System Design Study of the ALMA Future Correlator," *ASTRON-ALMA-FC-0002*, december 2000.

[12] *DSP-LINK3: Memory Mapped Expansion Bus Interface Specification*, SPECTRUM DSP System Worldwide, November 1998. Revision 1.02.

[13] John Reid Hauser, *Augmenting a Microprocessor with Reconfigurable Hardware.* University of California, Berkley, 2000.

[14] Stylianos Perissakis, *Balancing Computation and Memory in High Capacity Reconfigurable Arrays.* University of California, Berkley, 2000.

[15] *Stratix Devices: New Levels of System Integration*, Altera Corporation, April 2002. http://www.altera.com/products/devices/stratix/stx-index.jsp.

[16] *PMP8*, Signatec, 2002. http://www.signatec.com.

[17] G. Hampson, "Development of a Reconfigurable Digital Beamformer using Altera FPGAs and Mega Functions," *Internal ASTRON Report*, June 2000.

[18] K. van der Schaaf, "the TECH project," *ASTRON Internal Report*, November 1999.

[19] B. Morgan, "Learn to build a distributed Java applet that accesses server objects using CORBA," *http://www.javaworld.com/jw-10-1997/jw-10-corbajava.html*, April 2000.

Future Directions of (Programmable and Reconfigurable) Embedded Processors

Stephan Wong, Stamatis Vassiliadis, Sorin Cotofana

Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
{Stephan, Stamatis, Sorin}@CE.ET.TUDelft.NL

Abstract. *The advent of microprocessors in embedded systems has significantly contributed to the wide-spread utilization of embedded systems in our daily lives. Such embedded systems can be found in devices ranging from simple controllers found in power plants to sophisticated multimedia set-top boxes found in our homes. This is due to the fact that microprocessors, called embedded processors in this setting, are able to perform huge amounts of data processing required by embedded systems. In addition and equally important, embedded processors are able to achieve this at affordable prices. This has resulted in the fact that much effort must be placed in the design of embedded processors. In the last decade, we have been witnessing several changes in the embedded processors design fueled by two conflicting trends. First, the industry is dealing with cut-throat competition resulting in the need for increasingly faster time-to-market times in order to cut development costs. At the same time, embedded processors are becoming more complex due to the migration of increasingly more functionality to a single embedded processor in order to cut production costs. This has led to the quest for a flexible and reusable embedded processor which must still achieve high performance levels. As a result, embedded processors have evolved from simple microcontrollers to digital signal processors to programmable processors. We believe that this quest is leading to an embedded processor that comprises a programmable processor augmented with reconfigurable hardware. In this paper, we highlight several embedded processors characteristics and discuss how they have evolved over time when programmability and reconfigurability were introduced into the embedded processor design. Finally, we describe in-depth one possible approach that combines both programmability and reconfigurability in an integrated manner by utilizing microcode.*

1 Introduction

A technology turning point that made embedded consumer electronics systems an everyday reality has to be the advent of microprocessors. The technological developments that allowed single-chip processors (microprocessors) made the embedded systems inexpensive and flexible. Consequently, microprocessor-based embedded systems have been introduced into many new application areas. Currently, embedded programmable microprocessors in one form or another, from 8-bit micro-controllers to 32-bit digital signal processors and 64-bit RISC processors, are everywhere, in consumer electronic devices, home appliances, automobiles, network equipment, industrial control systems, etc. Interestingly, we are utilizing more than several dozens of embedded processors in our day-to-day lives without actually realizing it. For example, in modern cars such as the Mercedes S-class or the BMW 7-series, we can find over 60 embedded processors that control a multitude of functions, e.g., the fuel injection and the anti-lock braking system (ABS), that guarantee a smooth and foremost safe drive. The employment of embedded processors appear to grow in an exponential curve. Furthermore, it has been postulated [22] that the sales trend of embedded processors (microprocessors in this setting) will significantly outperform the sales of general-purpose PC processors.

In this positional paper, we describe several characteristics of embedded processors and investigate how these characteristics have changed over time driven by market requirements such as faster time-to-market times and development costs reductions. We will show that two strategies have been widely used to meet such market requirements, namely programmability and reconfigurability. Finally, we show a possible future direction in the embedded processor design that merges both strategies and thereby providing flexibility in both software and hardware design at the same time.

This paper is organized as follows. Section 2 introduces a general definition of embedded systems, discusses the characteristics of embedded systems that follow from the definition, and provides an in-depth discussion of traditional embedded processors characteristics. Section 3 discusses the need for programmability and several examples of such an approach. Section 4 discusses the use for reconfigurability and discusses how it affected the embedded processor's characteristics. Section 5 continues our discussion by describing what we think is the direction for future embedded processor that combines programmability and reconfigurability. Furthermore, we show an example of such an approach called the microcoded reconfigurable MOLEN embedded processor. Section 6 concludes this paper by stating several key observations in this paper.

2 Traditional Embedded Processor Characteristics

Embedded processors are a specific instance of embedded systems in general and therefore adhere to the characteristics of embedded systems. In this section, we provide a more traditional view on embedded processors by stating their characteristics deduced from our general definition of embedded systems:

Definition: *Embedded systems are (inexpensive) mass-produced elements of a larger system providing a dedicated, possibly time-constrained, service to that system.*

Before we highlight the main characteristics of embedded systems, we would like to comment on our one sentence definition of them. In most literature, the definition of embedded systems only states that they provide a dedicated service – the nature of the service is not relevant in this context – to a larger (embedding) system. However, we believe that all the issues related to the specification and design of embedded systems are very much anchored in the market reality. Consequently, in our opinion when we refer to embedded systems as mass-produced elements we draw the separation line between application-specific systems and embedded systems. We are aware that the separation line is quite thin in the sense that embedded systems are mostly indeed application-specific systems. However, we believe that low-production application-specific systems can not be considered as embedded systems, because they represent a niche market with very different set of requirements. For example, in low-production scenarios cost is usually not important while it is almost paramount for embedded systems to achieve low cost. Finally, we include the possibility for time-constrained behavior in our definition, because even if it is not characteristic to all the embedded systems it constitutes a particularity of a very large class of them, namely the real-time embedded systems.

The precise requirements of an embedded system is determined by its immediate environment. However, we still can classify the embedded system requirements in:

- **Functional requirements** are defined by the services that the embedded system has to perform for its immediate environment[1]. Such services usually include data gathering and some kind of data transformation/processing.
- **Temporal requirements** are the result of the time-constrained behavior of many embedded systems thereby introducing deadlines (explained later) for the service(s).
- **Dependability requirements** relates to the reliability, maintainability, and availability of the embedded system in question.

In the light of the previously stated embedded systems definition and requirements, we briefly point out what we think are the main characteristics of more traditional embedded processors and discuss in more detail the implications that these characteristics have on their specification and design processes. The first and probably the most important characteristic of embedded processors is that they are **application-specific**[2]. Given that the service (or application in processor terms) is known a priori, the embedded processor can be and should be optimized for its targeted application. In other words, embedded processors are definitely not general-purpose processors which are designed to perform reasonably for a much wider range of applications. Moreover, the fact that the application is known beforehand opens the road for *hardware/software co-design*, i.e., the cooperative and concurrent design of both hardware and software components of the processor. It is misleading to think that application-specific processors can not be programmed, because the signals controlling the processor can be perceived as rudimentary processor instructions, e.g., firmware or microcode [29], which could be re-arranged thus programmed. The hardware/software co-design style is very much particular to embedded processors and has the goal of meeting the processor level objectives by exploiting the synergism of hardware and software.

Another important characteristic of embedded processors is their **static structure**. When considering an embedded processor, the end-user has very limited access to software programming. The utilized software is provided by the processor integrator and/or application developer, resides on ROM memories, and is not visible to the end-user. The

[1] The immediate environment of an embedded systems can be either other embedded systems in the larger system or the world in which the larger system is placed.

[2] In accordance with our embedded systems definition, embedded processors are mass-produced application-specific processors. Therefore, we consider graphics processors in game consoles to be embedded processors. On the other hand, graphics processors intended for military simulators are not since they are not mass-produced.

end-user can not change nor reprogram the basic operations of the embedded processor, but he is usually allowed to program a (different) sequence of basic operations.

Embedded processors are essentially non-homogeneous processors and this characteristic is induced by the **heterogeneous** character of the process within which the processor is embedded. Designing a typical embedded processor does not only mix hardware design with software design, but it also mixes design styles within each of these categories. To put more light on the heterogeneity issue, we depicted in Figure 1 (from [16]) an example signal processing embedded processor. The heterogeneous character can be seen in many aspects of the embedded processor design as follows:

– both analog and digital sub-processors may be present in the system;
– the hardware may include microprocessors, microcontrollers, digital signal processors (DSPs), application-specific integrated circuits (ASICs);
– the topology of the system is rather irregular;
– various software modules as well as a multitasking real-time operating system.

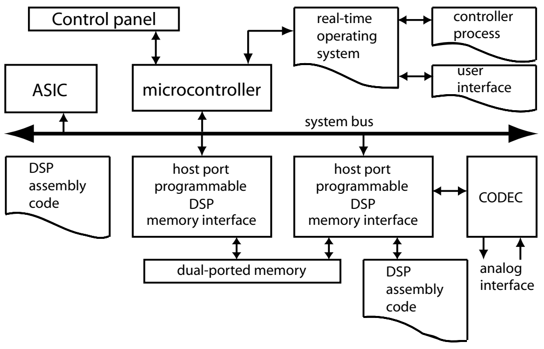


Fig. 1. Signal Processing Embedded Processor Example (from [16]).

Generally speaking, the intrinsic heterogeneity of embedded processors largely contributes to the overall complexity and management difficulties of the design process. However, one can say that heterogeneity is in the case of embedded processors design a necessary evil. It provides better design flexibility by providing a wide range of design options. In addition, it allows each required function to be implemented on the most adequate platform that is deemed necessary to meet the posed requirements.

Embedded processors are also **mass-produced** elements separating them from (low-production) application-specific processors. This characteristic imposes a different set of requirements for the embedded processor design, because embedded processor vendors face fierce competition in order to gain more market capitalization. An example requirement involves the cost/performance sensitiveness of embedded processors making low cost almost always an issue. Other related design issues include: high-production volume, small time-to-market window, and fast development cycles.

A large number of embedded processors performs **real-time** processing introducing the notion of *deadlines*. Roughly speaking, deadlines can be classified in: hard real-time deadlines and soft real-time deadlines. Missing hard deadline can be catastrophic while missing soft deadline only results in some non-fatal glitches. Both types of deadlines are known a priori much like that the functionality is known beforehand. Therefore, deadlines determine the minimum level of performance that must be achieved. When facing hard deadlines, special attention must also be paid to other systems connected to the embedded processor since they can negatively influence its behavior.

3 The Need for Programmability

In the early nineties, we were witnessing a trend in the embedded processors market that was reshaping the characteristics of traditional embedded processors as introduced in Section 2. Driven by market forces, the lengthy embedded processors design cycles had to be shortened in order to keep up with or stay in front of competitors. In addition, production and development costs had to be reduced in order to stay competitive. By highlighting the traditional embedded processors design, we discuss ”large scale” programmability[3] which has been used to address these two issues.

The heterogeneity of the embedded systems demanded a multitude embedded processors to be designed for a single system. This was further strengthened by the disability of the semiconductor technology at that time to produce large chips. As a result, the multitude of embedded processors requires lengthy design and verification times, especially for their interfaces. On the other hand, subsequent design cycles could be significantly reduced if only a small number of the embedded processors requires redesign. This delicate balance between long initial design cycles and possibly shortened subsequent design cycles was disturbed when advancing semiconductor technology allowed increasingly more gates to be put on a single chip. Fueled by the need to incorporate increasingly more functionality into (in order to distinguish yourself from competitors) and to decrease the cost of embedded systems, the functionalities of embedded processors were expanded. More complex and larger embedded processors did not decrease the initial design cycles. However, the subsequent redesign cycles were increased, because we are dealing with highly optimized circuits meaning that subsequent designs are not necessarily easier than the initial ones.

In the search for design flexibility in order to decrease design cycles and reduce subsequent design costs, functions were separated into time-critical functions and non-time-critical ones. One could say that the embedded processors design paradigm has shifted from one that is based on the functional requirements to one that is based on the temporal requirements. The collection of non-’time-critical’ functions could then be performed on a single chip[4]. The remaining time-critical function are to be implemented in high-speed circuits achieving maximum performance. The main benefit of this approach is that the large (possibly slower) chip can be reused in subsequent designs resulting in shorter design cycles. Moreover, the large chip also exhibits a more general-purpose behavior and its design becomes more like the design of general-purpose pro-

[3] One could argue that programmability has always been part of embedded processors. However, programmability introduced in this section significantly differs from the limited (low-level) programmability of traditional embedded processors.

[4] Possibly implemented in a slower technology in order to reduce cost.

cessors. The design of general-purpose processors can be divided into three distinct fields [14]: architecture[5], implementation, and realization.

In Section 2, we stated that more traditional embedded processors are application-specific and static in nature. However, in this section we also stated that increasingly more functionality is embedded into a single embedded processor. Is such a processor still application-specific and can we still call such a processor an embedded processor? The answer to this question is affirmative since such a processor is still embedded if the other constraints (mass-produced, providing a dedicated service, etc.) are observed. Given that increasing functionality usually implies more exposure of the processor to the programmer, embedded processor have become indeed less static as they can now be reused for other applications areas due to their programmability. In this light, two scenarios in the design of programmable embedded processors can be distinguished:

– **Adapt an existing general-purpose architecture** and implement such an architecture. This scenario reduces development costs albeit such architectures must be licensed. Furthermore, since such architectures were not adapted to embedded processors still some development times is needed to modify such architectures.
– **Build a new embedded processor architecture** from scratch. In this scenario, the embedded processor development takes longer, but the final architecture is more tuned towards the specific application the embedded processor is intended for.

Several examples of the first scenarios can be found. A well-known example is the MIPS architecture [4]. In this case, the architecture has been adapted towards embedded processors by MIPS Technologies, Inc. which develops the architecture separately from other embedded systems vendors. Another well-known example is the ARM architecture [1]. It is a RISC architecture that was firstly intended for low-power PCs (1987), but has been quickly adapted to become an embeddable RISC core (1991). Since then the ARM architecture has been subject to numerous modifications and/or extensions in order to optimize it for targeted applications. A well-known implementation is the StrongARM core which was jointly developed by then Digital Semiconductor and ARM Ltd. This core was intended to provide high performance at extreme low-power. The most current implementation of this core developed by Intel Corp. is called the Intel PCA Application Processor [3] intended for PDA handhelds. Other example general-purpose architectures that have been adapted include: IBM PowerPC [2], Sun UltraSPARC [8], the Motorola 68000/Coldfire [5], and many more. An example of the second scenario is the Trimedia VLIW architecture [9] from Trimedia technologies, Inc. which was originally developed by Philips Electronics. Its application area was multimedia processing and can now be found in many televisions, digital receivers, and other digital video editing boards. Figure 2 shows a block diagram of the Trimedia TM-1300 processor. It contains a VLIW processor core that controls the other specialized hardware cores and performs other functions that do not need real-time performance.

Summarizing, the characteristics mentioned in Section 2 can be easily reflected in the three processor design stages (architecture, implementation, and realization). The characteristic of embedded processors being application-specific is exhibited by the

[5] The architecture of any computer system is defined to be the conceptual structure and functional behavior as seen by its immediate user.

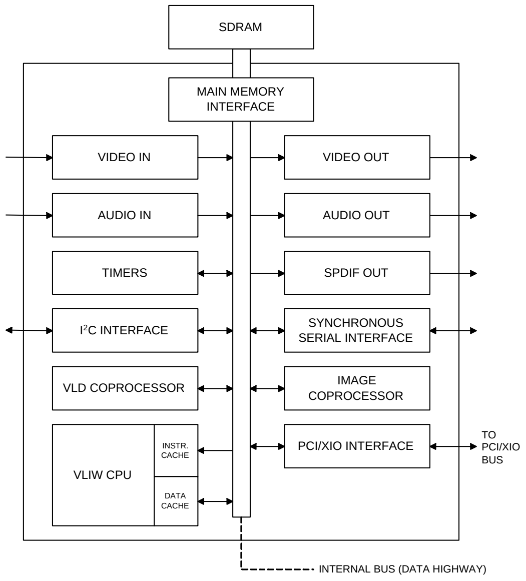


Fig. 2. The Trimedia TM-1300.

fact that the architecture only contains those operations that really need support from the applications set. The static structure characteristic exhibits itself by having a fixed architecture, a fixed implementation, and a fixed realization. The heterogeneity characteristic exhibits itself by the utilization of programmable processor core with other specialized hardware units. In addition, a multitude of different functional units exist on the programmable processor core. The mass-produced characteristic is exhibiting itself in the realization process by only utilizing proven technology that is therefore cheap and reliable. The requirement of real-time processing exhibits itself by requiring architectural support for frequently used operations, extensively parallel (if possible) implementations, and realization incorporating high-speed components.

Finally, a wide variety of design issues also have their impact on the architecture, implementation, and realization of an embedded processor. However, due to the vast variety of design issues, such as cost, performance, cost/performance ratio, high/low production volume, fast development, and small time-to-market windows, we refrain ourselves from discussing these issues in the light of architecture, implementation, realization of embedded processors. However, it must be clear that each design issue has a certain level of impact on the architecture, implementation, and realization of an embedded processor.

4 Early Time Reconfigurability

In the mid-nineties, we were witnessing a second trend in the embedded processors design that was reshaping the design methodology of embedded processors and consequently redefined some of their characteristics. Previously, in the design of embedded processors application-specific integrated circuits (ASICs) were still commonplace and the design of ASICs required lengthy design cycles. It requires several roll-outs of the embedded processor chips in question in order to test/verify all the functional, temporal, and dependability requirements. Therefore, design cycles of 18 months or longer were commonplace rather than exceptions. A careful step towards reducing such lengthy design cycles is to use reconfigurable hardware, also referred to as fast prototyping. This allows embedded processor designs to be mapped early on in the design cycle to reconfigurable hardware, in particular field-programmable gate arrays (FPGAs), enabling early functionality testing and thereby reducing the number of chip roll-outs. However, such hardware initially were limited in size and therefore only small parts of embedded processor designs could be tested. Consequently, still roll-out(s) of the complete chip (implemented in ASICs) were still required in order to test the overall functionality.

In recent years, the reconfigurable technology has progressed in a fast pace and it has currently arrived at the point that embedded processor designs requiring million(s) of gates can be implemented on such structures. In addition, the performance gap that existed between FPGAs and ASICs is rapidly decreasing. This development in technology has also changed the role of reconfigurable hardware in embedded processors design. Instead of only serving fast prototyping purposes, embedded processors implemented in reconfigurable hardware are actually being shipped in final products. An additional benefit of this development is that bugs found in such embedded processors can be easily rectified resulting in much higher user satisfaction. Furthermore, design improvements can also be easily incorporated during maintenance sessions. In the following, we revisit the embedded processor characteristics mentioned in Section 2 and investigate whether they still hold in case embedded processors are build using FPGAs.

application-specific Embedded processors built utilizing reconfigurable hardware are still application-specific in the sense that the implementations are still targeting such applications. Utilizing such implementations for other purposes will prove to be very hard or it will not achieve the required performance levels.

static structure This characteristic has been affected the most by the utilization of reconfigurable hardware. From a pure technical perspective, the structure of a reconfigurable embedded processor is not static since its functionality can be changed, either during maintenance or during operation. However, we have to consider the frequency of this happening. In most cases, an implementation is chosen for the reconfigurable embedded processor and it is not changed anymore between maintenance intervals. Therefore, from the user's perspective the structure of the embedded processor is still static. In the next section, we will explore the possibility that the functionality of an embedded processor needs to be changed even during operation.

heterogeneous This characteristics is still very much present in the case of reconfigurable embedded processors. We have added an additional technology into the mix in which embedded processors can be realized. For example, the latest FPGA offering from both Altera Inc. (Stratix [7]) and Xilinx Inc. (Virtex II [10]) integrates on a single chip the following: memory, logic, I/O controllers, and DSP blocks.

mass-produced This characteristic is still applicable to reconfigurable hardware. Early on, reconfigurable hardware has only been used to verify the functionality of design and therefore were not implemented in actual shipped embedded processors. As the technology progressed, it allowed reconfigurable hardware to be produced at much lower costs and therefore opening the possibility of actually shipping reconfigurable hardware in actual products. This is actually the case at this moment.

real-time In the beginning, we were witnessing the incorporation of reconfigurable hardware only for non-'time-critical' functions. As the technology of reconfigurable continue to progress and making reconfigurable hardware much faster, we are also witnessing their incorporation in actual products where real-time performance is required, such as multimedia decoders.

5 Future Embedded Processors

In Sections 3 and 4, we have shown that both programmability and reconfigurability have been introduced into the embedded processor design trajectory born out of the need to reduce design cycles and reduce development costs. Programmability allows the utilization of high-level programming languages (like C) and thereby easing application development. Reconfigurability allows designs to be tested early on in terms of functionality and diminishes the need for expensive chip roll-outs. The merging of both strategies in the embedded processor design (if possible) will result in two main advantages. First, the design flexibility is hugely increased, because it allows easy design space exploration in both software and hardware. Second, it allows rapid application development since the software and hardware can be realized utilizing high-level programming and hardware description languages. When correctly incorporated, the combination of programmability and reconfigurability allows embedded processors to change their functionality dynamically during operation (in run-time).

The mentioned advantages and enabling FPGA technologies have even resulted in that programmable processor cores are under consideration to be implemented in the same FPGA structures, e.g., Nios from Altera [6] and MicroBlaze from Xilinx [11]. However, the utilization of programmable embedded processors that are augmented with reconfigurable hardware also poses several issues that must be addressed:

– **Long reconfiguration latencies:** When considering dynamic run-time reconfigurations, such latencies may greatly penalize the performance, because any computation must be halted until the reconfiguration has finished.
– **Limited opcode space:** The initiation and control of the reconfiguration and execution of various implementations on the reconfigurable hardware require the introduction of new instructions. This puts much strain on the opcode space.
– **Complicated decoder hardware:** The multitude of newly introduced instructions greatly increased the complexity of the decoder hardware.

In the following, we discuss one possible approach [28] (introduced by us) in merging programmability with reconfigurability in the design of embedded processors. The approach utilizes microcode to alleviate the mentioned problems. Microcode consists of a sequence of (simple) microinstructions that, when executed in a certain order, performs "complex" operations. This approach allows "complex" operations to be performed on much simpler hardware. In this section, we consider the reconfiguration (either off-line or run-time) and execution processes as complex operations. The main benefits of our approach can be summarized as follows:

– **Reduced reconfiguration latencies:** Microcode used to control the reconfiguration process allows itself to be cached on-chip. This results in faster access times to the reconfiguration microcode and thus in turn reduces the reconfiguration latencies.
– **Reduced opcode space requirements:** By only pointing to microcode (explained later), we only require (at most) three new instructions and not separate instructions for each and every supported operation.
– **Reduced decoder hardware complexity:** Due to the inclusion of only a few instructions, complex instruction decoding hardware is no longer required.

In Section 5.1, we revisit microcode from its beginnings to its current implementation within a high-level microprogrammed machine. In Section 5.2, we discuss in-depth our proposed MOLEN embedded processor. Finally, in Section 5.3, we briefly highlight several other approaches in this field that are comparable in one way or another.

5.1 Revisiting Microcode

Microcode, introduced in 1951 by Wilkes [29], constitutes one of the key computer engineering innovations. Microcode de facto partitioned computer engineering into two distinct conceptual layers, namely: architecture and implementation. This is in part because emulation allowed the definition of complex instructions which might have been technologically not implementable (at the time they were defined), thus projecting an architecture to the future. That is, it allowed computer architects to determine a technology-independent functional behavior (e.g., instruction set) and conceptual structures providing the following possibilities:

– Define the computer's architecture as a programmer's interface to the hardware rather than to a specific technology dependent realization of a specific behavior.
– Allow a single architecture to be determined for a “family” of implementations giving rise to the important concept of compatibility. Simply stated, it allowed programs to be written for a specific architecture once and run at “infinitum” independent of the implementations.

Since its beginnings, as introduced by Wilkes, microcode has been a sequence of micro-operations (microprogram). Such a microprogram consists of pulses for operating the gates associated with the arithmetical and control registers. Figure 3 depicts the method of generating this sequence of pulses. First, a timing pulse initiating a micro-operation enters the decoding tree and depending on the setup register R, an output is generated. This output signal passes to matrix A which in turn generates pulses to control arithmetical and control registers, thus performing the required micro-operation. The output signal also passes to matrix B, which in its turn generates pulses to control the setup register R (with a certain delay). The next timing pulse will therefore generate the next micro-operation in the required sequence due to the changed register R.

Over the years, the Wilkes' model has evolved into a high-level microprogrammed machine as depicted in Figure 4[6]. In this figure, the control store contains microinstructions (representing one or more micro-operations) and the sequencer determines

[6] The memory address register (MAR) is used to store the memory address in the main memory from which data must be loaded of to which data is stored. The memory data register (MDR) stores the data that is communicated to or from the main memory.

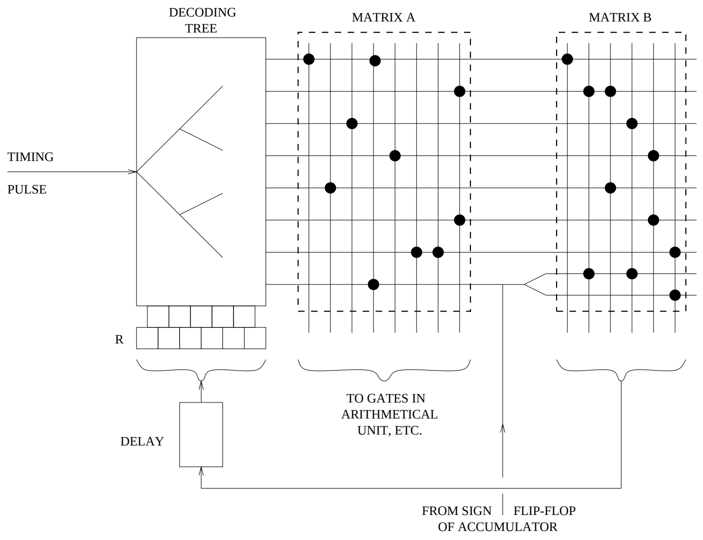

DECODING TREE
MATRIX A
MATRIX B
TIMING
PULSE
R
DELAY
TO GATES IN ARITHMETICAL UNIT, ETC.
FROM SIGN FLIP-FLOP OF ACCUMULATOR

Fig. 3. Wilkes' microprogram control model [29].

the next microinstruction to execute. The control store and the sequencer correspond to Wilkes' matrices A and B respectively. The machine's operation is as follows:

1. The control store address register (CSAR) contains the address of the next microinstruction located in the control store. The microinstruction located at this address is then forwarded to the microinstruction register (MIR).
2. The microinstruction register (MIR) decodes the microinstruction and generates smaller micro-operation(s) accordingly that need to be performed by the hardware unit(s) and/or control logic.
3. The sequencer utilizes status information from the control logic and/or results from the hardware unit(s) to determine the next microinstruction and stores its control store address in the CSAR. It is also possible that the previous microinstruction influences the sequencer's decision regarding which microinstruction to select next.

It should be noted that in microcoded engines not all instructions access the control store. As a matter of fact, only emulated instructions have to go through the microcode logic. All other instructions will be executed directly by the hardware (following path (α) in Figure 4). That is, a microcoded engine is as a matter of fact a hybrid of the implementation having emulated instructions and hardwired instructions[7].

5.2 Microcoded Reconfigurable MOLEN Embedded Processor

In this section, only a brief description of the MOLEN embedded processor is given, We refer to [28] for a more detailed description. In its more general form, the proposed machine organization can be described as in Figure 5. In this organization, the

[7] That is, contrary to some believes, from the moment it was possible to implement instructions, microcoded engines always had a hardwired core that executed RISC instructions.

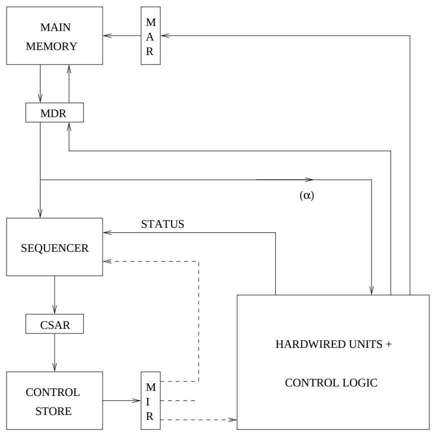


Fig. 4. A high-level microprogrammed machine.

I_BUFFER stores the instructions that are fetched from the memory. Subsequently, the ARBITER performs a partial decoding on these instructions in order to determine where they should be issued. Instructions that have been implemented in fixed hardware are issued to the core processing (CP) unit which further decodes them before sending them to their corresponding functional units. The needed data is fetched from the general-purpose registers (GPRs) and results are written back to the same GPRs. The control register (CR) stores other status information.

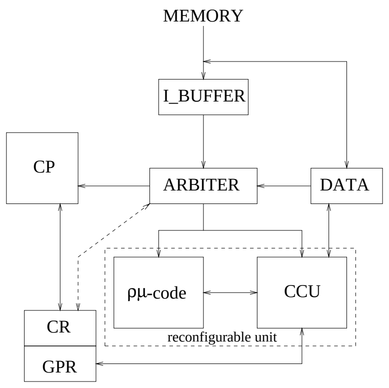


Fig. 5. The proposed machine organization.

The reconfigurable unit consists of a custom configured unit (CCU)[8] and the $\rho\mu$-code unit. An operation[9] performed by the reconfigurable unit is divided into two dis-

[8] Such a unit could be for example implemented by a Field-Programmable Gate Array (FPGA).
[9] An operation can be as simple as an instruction or as complex as a piece of code of a function.

tinct process phases: **set** and **execute**. The **set** phase is responsible for configuring the CCU enabling it to perform the required operation(s). Such a phase may be subdivided into two sub-phases: partial **set** (*p*-**set**) and complete **set** (*c*-**set**). The *p*-**set** sub-phase is envisioned to cover common functions of an application or set of applications. More specifically, in the *p*-**set** sub-phase the CCU is *partially* configured to perform these common functions. While the *p*-**set** sub-phase can be possibly performed during the loading of a program or even at chip fabrication time, the *c*-**set** sub-phase is performed during program execution. In the *c*-**set** sub-phase, the remaining part of the CCU (not covered in the *p*-**set** sub-phase) is configured to perform other less common functions and thus *completing* the functionality of the CCU. The configuration of the CCU is performed by executing reconfiguration microcode[10] (either loaded from memory or resident) in the $\rho\mu$-code unit. In the case that partial reconfigurability is not possible or not convenient, the *c*-**set** sub-phase can perform the entire configuration. The **execute** phase is responsible for actually performing the operation(s) on the (now) configured CCU by executing (possibly resident) execution microcode stored in the $\rho\mu$-code unit.

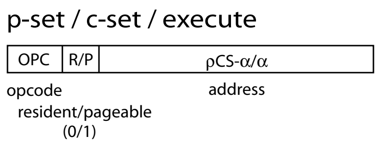


Fig. 6. The *p*-**set**, *c*-**set**, and **execute** instruction formats.

In relation to these three phases, we introduce three new instructions: *c*-**set**, *p*-**set**, and **execute**. Their instruction format is given in Figure 6. We must note that these instructions do *not* specifically specify an operation and then load the corresponding reconfiguration and execution microcode. Instead, the *p*-**set**, *c*-**set**, and **execute** instructions directly point to the (memory) location where the reconfiguration or execution microcode is stored. In this way, different operations are performed by loading different reconfiguration and execution microcodes. That is, instead of specifying new instructions for the operations (requiring instruction opcode space), we simply point to (memory) addresses. The location of the microcode is indicated by the resident/pageable-bit (R/P-bit) which implicitly determines the interpretation of the address field, i.e., as a memory address α (R/P=1) or as a ρ-CONTROL STORE address ρCS-α (R/P=0) indicating a location within the $\rho\mu$-code unit. This location contains the first instruction of the microcode which must always be terminated by an *end_op* microinstruction.

The $\rho\mu$-code unit: The $\rho\mu$-code unit can be implemented in configurable hardware. Since this is only a performance issue and not a conceptual one, it is not considered further in detail. In this presentation, for simplicity, we assume that the $\rho\mu$-code unit is hardwired. The internal organization of the $\rho\mu$-code unit is given in Figure 7. In all phases, microcode is used to perform either reconfiguration of the CCU or control the execution on the CCU. Both types of microcode are conceptually the same and no distinction is made between them in the remainder of this section. The $\rho\mu$-code unit comprises two main parts: the SEQUENCER and the ρ-CONTROL STORE. The

[10] Reconfiguration microcode is generated by translating a reconfiguration file into microcode.

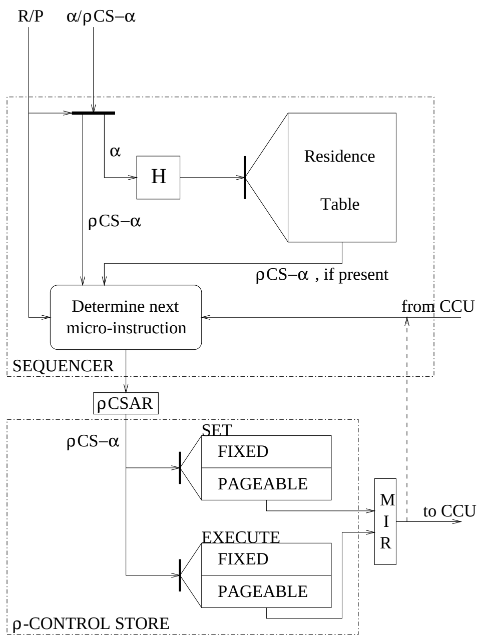


Fig. 7. $\rho\mu$-code unit internal organization.

SEQUENCER mainly determines the microinstruction execution sequence and the ρ-CONTROL STORE is mainly used as a storage facility for microcodes. The execution of microcodes starts with the SEQUENCER receiving an address from the ARBITER and interpreting it according to the R/P-bit. When receiving a memory address, it must be determined whether the microcode is already cached in the ρ-CONTROL STORE or not. This is done by checking the RESIDENCE TABLE which stores the most frequently used translations of memory addresses into ρ-CONTROL STORE addresses and keeps track of the validity of these translations. It can also store other information: least recently used (LRU) and possibly additional information required for virtual addressing[11] support. In the cases that a ρCS-α is received or a valid translation into a ρCS-α is found, it is transferred to the 'determine next microinstruction'-block. This block determines which (next) microinstruction needs to be executed:

– When receiving address of first microinstruction: Depending on the R/P-bit, the correct ρCS-α is selected, i.e., from instruction field or from RESIDENCE TABLE.
– When already executing microcode: Depending on previous microinstruction(s) and/or results from the CCU, the next microinstruction address is determined.

The resulting ρCS-α is stored in the ρ-control store address register (ρCSAR) before entering the ρ-CONTROL STORE. Using the ρCS-α, a microinstruction is fetched from the ρ-CONTROL STORE and then stored in the microinstruction register (MIR) before it controls the CCU reconfiguration or before it is executed by the CCU.

[11] For simplicity of discussion, we assume that the system only allows real addressing.

The ρ-CONTROL STORE comprises two sections[12], namely a **set** section and an **execute** section. Both sections are further divided into a **fixed** part and **pageable** part. The fixed part stores the resident reconfiguration and execution microcode of the **set** and **execute** phases, respectively. Resident microcode is commonly used by several invocations (including reconfigurations) and it is stored in the fixed part so that the performance of the **set** and **execute** phases is possibly enhanced. Which microcode resides in the fixed part of the ρ-CONTROL STORE is determined by performance analysis of various applications and by taking into consideration various software and hardware parameters. Other microcodes are stored in memory and the pageable part of the ρ-CONTROL STORE acts like a cache to provide temporal storage. Cache mechanisms are incorporated into the design to ensure the proper substitution and access of the microcode present in the ρ-CONTROL STORE.

5.3 Other reconfigurability approaches

In the previous section, we have introduced a machine organization where the hardware reconfiguration and the execution on the reconfigured hardware is done in firmware via the ρ-microcode (an extension of the classical microcode to include reconfiguration and execution for resident and non-resident microcode). The microcode engine is extended with mechanisms that allow for permanent and pageable reconfiguration and execution microcode to coexist. We also provide partial reconfiguration possibilities for “off-line” configurations and prefetching of configurations. Regarding related work we have considered more than 40 machine proposals. We report here a number of them that somehow use some partial or total reconfiguration prefetching. It should be noted that our scheme is rather different in principle from all related work as we use microcode, pageable/fixed local memory, hardware assists for pageable reconfiguration, partial reconfigurations, etc.. As it will be clear from the short description of the related work, we differentiated from them in one or more mechanisms.

The *Programmable Reduced Instruction Set Computer (PRISC)* [25] attaches a Programmable Functional Unit (PFU) to the register file of a processor for application-specific instructions. Reconfiguration is performed via exceptions. In an attempt to reduce the overhead connected with FPGA reconfiguration, Hauck proposed a slight modification to the PRISC architecture in [20]: an instruction is explicitly provided to the user that behaves like a NOP if the required circuit is already configured on the array, or is in the process of being configured. By inserting the configuration instruction before it is actually required, a so-called *configuration prefetching* procedure is initiated. At this point the host processor is free to perform other computations, overlapping the reconfiguration of the PFU with other useful work. The *OneChip* introduced by Wittig and Chow [30] extends PRISC and allows PFU for implementing any combinational or sequential circuits, subject to its size and speed. The system proposed by Trimberger [27] consists of a host processor augmented with a PFU, *Reprogrammable Instruction Set Accelerator* (RISA), much like the PRISC mentioned above. Concerning the management and control of the reprogramming procedure, Trimberger mentions that the RISA reconfiguration is under control of a hardwired execution unit. However, it is

[12] Both sections can be identical, but are probably only differing in microinstruction wordsizes.

not obvious if an explicit SET instruction is available. The *Reconfigurable Multimedia Array Coprocessor* (REMARC) proposed by Miyamori and Olukotun [24] augments the instruction set of a MIPS core. As the coprocessor does not have a direct access to the main memory, the host processor has to write the input data to the coprocessor data registers, initiate the execution, and finally read the results from the coprocessor data registers. An explicit reconfiguration instruction is provided. *Garp* designed by Hauser and Wawrzynek [21] is another example of a MIPS derived Custom Computing Machine (CCM). The FPGA-based coprocessor has a direct access to the standard memory. The MIPS instruction set is augmented with several non-standard instructions dedicated to loading a new configuration, initiating the execution of the newly configured computing facilities, moving data between the array and the processor's own registers, saving/retriving the array states, branching on conditions provided by the array, etc. The coprocessor is aimed to run autonomously with the host processor. In the *OneChip-98* introduced by Jacob and Chow[23], the computing resources are loaded *on-demand* when a miss is detected. *Alternatively*, the resources are *pre-loaded* by using compiler directives. Several comments regarding these assertions are worth to be provided. If an on-demand loading strategy is employed, then the user has no control on the reconfiguration procedure. In the pre-loading strategy, an explicit reconfiguration instruction is provided to the user and the reconfiguration procedure is indeed under the control of the user. PRISM (*Processor Reconfiguration Through Instruction-Set Metamorphosis*) one of the earliest proposed CCM [12][13], was developed as a proof-of-concept system, in order to handle the loading of FPGA configurations, the compiler inserts library function calls into the program stream [13]. From this description, we can conclude that an explicit reconfiguration procedure is available. Gilson [17] CCM architecture consists of a host processor and two or more FPGA-based *computing devices*. The host controls the reconfiguration of FPGAs by loading new configuration data through a Host Interface into the FPGA Configuration Memory. The reconfiguration process can be performed such that when one computing device is being reconfigured and, therefore, is idle, the others continue executing. The write into the configuration memory instruction can play the role of an explicit reconfiguration instruction. Therefore, a *pre-loading* strategy is employed. Schmit [26] proposes a partial run-time reconfiguration mechanism, called *pipeline reconfiguration* or *striping*, by which the FPGA is reconfigured at a granularity that corresponds to a pipeline stage of the application being implemented. An application which has been broken up into pipeline stages can be mapped to a striped FPGA. The pipeline stages are known as *stripes*; the stages of the application are called *virtual stripes*, and the hardware stages which the virtual stages are loaded into are called *physical stripes*. The PipeRench coprocessor developed by a team with Carnegie Mellon University [15][18] is focused on implementing linear (1-D) pipelines of arbitrary length. PipeRench is envisioned as a coprocessor in a general-purpose computer, and has direct access to the same memory space as the host processor. The virtual stripes of the application are stored into an on-chip configuration memory. A single physical stripe can be configured in one read cycle with data stored in such a memory. The configuration of a stripe takes place concurrently with execution of the other stripes. The *Reconfigurable Data Path Architecture* (rDPA) is also a self-steering autonomous reconfigurable architecture. It consists of a mesh of identical Data Path Units (DPU)[19].

The data-flow direction through the mesh is only from west and/or north to east and/or south and is also data-driven. A word entering rDPA contains a configuration bit which is used to distinguish the configuration information from data. Therefore, a word can specify either a SET or an EXECUTE instruction, the arguments of the instructions being the configuration information or data to be processed. A set of computing facilities can be configured on rDPA.

6 Conclusions

In this positional paper, we have described several characteristics of embedded processors that were logically deduced from embedded systems characteristics in general. Driven by market requirements, two strategies were followed in order to reduce design cycles and development costs. First, programmability was introduced as a means to combine all non-'time-critical' functions to be performed by a 'general-purpose'-like embedded processor. Such an embedded processor could then be reused in subsequent design and thereby greatly reducing design cycles. Second, reconfigurability was initially only utilized as fast prototyping. Over time, technological advances in reconfigurable hardware in terms of size and performance have led to the fact the reconfigurable embedded processors are actually incorporated in shipped embedded systems. We believe that the future of embedded processors design lies in the merging of both strategies. Programmability allows the utilization of high-level programming languages (like C) and thereby easing application development. The utilization of reconfigurable hardware combines design flexibility and fast prototyping. At the same time, the processing performance of reconfigurable hardware is nearing that of application-specific integrated circuits. Finally, in this paper we have highlighted one possible framework in which future embedded processor design can be performed. The proposed MOLEN embedded processor combines software programming (by utilizing a programmable processor core) with hardware programming (utilizing microcode to control the reconfigurable hardware). Such an approach provides possibilities in combatting several issues associated with reconfigurable hardware.

References

1. ARM architecture. http://www.arm.com.
2. IBM PowerPC. http://www-3.ibm.com/chips/products/powerpc/.
3. Intel PCA Application Processors. http://www.intel.com/design/pca/applications processors/index.htm.
4. MIPS architecture from MIPS Technologies. http://www.mips.com.
5. Motrola 68000/Coldfire Family. http://e-www.motorola.com/webapp/sps/site/homepage.jsp?nodeId=03M0ylgrpxN.
6. Nios Embedded Processor. http://www.altera.com/products/devices/excalibur/exc-nios_index.html.
7. Stratix Family. http://www.altera.com/products/devices/stratix/stx-index.jsp.
8. Sun UltraSPARC IIe. http://www.sun.com/microelectronics/UltraSPARC-IIe/index.html.
9. Trimedia VLIW architecture. http://www.trimedia.com.

10. Virtex-II 1.5V FPGA Family: Detailed Functional Description . http://www.xilinx.com/partinfo/databook.htm.
11. Xilinx MicroBlaze. http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=microblaze.
12. P.M. Athanas. *An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*. PhD thesis, Brown University, Providence, Rhode Island, May 1992.
13. P.M. Athanas and H.F. Silverman. Processor Reconfiguration through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
14. G.A. Blaauw and F.P. Brooks. *Computer Architecture: Concepts and Evolution*. Addison-Wesley, 1997.
15. S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D.E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *6th International Symposium on Field Programmable Gate Arrays*, pages 55–64, California, USA, 1998.
16. W.-T. Chang, A. Kalavade, and E.A. Lee. Effective Heterogeneous Design and Co-Simulation. In Giovanni de Michelli and Mariagiovanna Sami, editors, *Hardware/Software Co-Design*, pages 187–211. Kluwer Academic Publishers, 1995.
17. K.L. Gilson. Integrated Circuit Computing Device Comprising a Dynamically Configurable Gate Array Having a Microprocessor and Reconfigurable Instruction Execution Means and Method Therefore. U.S. Patent No. 5,361,373, November 1994.
18. S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *The 26th International Symposium on Computer Architecture*, pages 28–39, Georgia, USA, May 1999.
19. R.W. Hartenstein, R. Kress, and H. Reinig. A New FPGA Architecture for Word-Oriented Datapaths. In *Field-Programmable Logic: Architectures, Synthesis and Applications. 4th International Workshop on Field-Programmable Logic and Applications*, Lecture Notes in Computer Science, pages 144–155, Czech Republic, September 1994.
20. S.A. Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. In *6th International Symp. on Field Programmable Gate Arrays*, pages 65–74, California, 1998.
21. J.R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symp. on FPGAs for Custom Computing Machines*, pages 12–21, California, 1997.
22. J. Hennessy. The Future of Systems Research. *Computer*, pages 27–33, 1999.
23. J.A. Jacob and P. Chow. Memory Interfacing and Instruction Specification for Reconfigurable Processors. In *ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 145–154, Monterey, California, 1999.
24. T. Miyamori and K. Olukotun. A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 2–11, California, 1998.
25. R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, Cambridge, Massachusetts, May 1994.
26. H. Schmit. Incremental Reconfiguration for Pipelined Applications. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, California, April 1997.
27. S.M. Trimberger. Reprogrammable Instruction Set Accelerator. U.S. Patent No. 5,737,631, April 1998.
28. S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-Coded Processor. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL2001)*, pages 275–285, 2001.
29. M. V. Wilkes. The Best Way to Design an Automatic Calculating Machine. In *Report of the Manchester University Computer Inaugural Conference*, pages 16–18, July 1951.
30. R.D. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, 1996.

A high level synthesis framework for reconfigurable architectures

Loïc Lagadec, Bernard Pottier, Oscar Villellas-Guillen

Architectures et Systèmes,
Université de Bretagne Occidentale
UFR Sciences, 20 av. LeGorgeu, Brest 29285, France

Abstract. The Madeo framework is divided in three parts for low level architecture modelization and tools, high level logic synthesis, and computation modeling and synthesis. This paper describes a compilation technique used in the second part to allow programming of arbitrary primitives associated to object classes.
The compiler handles methods associated to set or interval of values. It produces a network of lookup-tables equivalent to the function calls that are translated into logic tables and mapped to a target reconfigurable architecture.
A first effect of this technique is to reduce the logic complexity mostly at high level taking advantage of sparse set of values. A second property is that high level compilation techniques can be applied on expressions independently of language semantics. Detailed results are given in the case of a Reed-Solomon RAID coder/decoder generator.

1 Introduction

General context

Integration technology is providing hardware resources at high rate while the hardware development methods are progressing slowly. This can be seen as the repetition of a common situation where a mature technical knowledge is providing useful possibilities in excess of current method capabilities. An answer is to change the development process in order to avoid work repetition and to provide more productivity by secure assembly of standard components.

This situation is also known from computer scientists since it has been encountered earlier in the programming language story[DC90]. The first ages of this story are: (1) symbolic expression of computations (Fortran), (2) structured programs (Algol, Pascal), (3) modularity, and code abstraction via interfaces and hiding (Ada, Modula2, object oriented programming).

Modularity came at the age where efficient engineering of large programs was the main concern, and when the task of programming was overtaken by the organization problems. *System development* can be considered as a new age for computer architecture design, with hardware description languages needing to be transcended by higher level of descriptions to increase productivity.

Companies developing applications have their specific methods for *design* and *production management* in which they can represent their products, tools and hardware or software components. The method that ensure the feasibility of a product is leading technical choices and developments. It also change some of the rules in design since most of the application is achieved in a top-down fashion using components or code generators reaching the functional requirements.

Reconfigurable architectures

FPGAs are one of the driving forces for integration technology progresses, due to their increasing field of applications.

Like software and hardware programming languages, reconfigurable architectures are sensitive to scale mutations. As the chip size is increasing, the characteristics of the application architecture change, with new needs for structured communications, more efficiency on arithmetic operators., and partial reconfigurability. The software follows slowly, migrating from HDL to HLL. Preserving the developments and providing a sane support for production tools is, in our opinion, a major issue.

Madeo

MADEO is a medium term project that make use of open object modeling to provide access to hardware resources and code portability on reconfigurable architectures.

The project structure has three parts that interact closely (bottom-up):

1. **Reconfigurable architecture model and its associated generic tools.** The representation of practical architectures on a generic model enables sharing of basic tools such as place and route, allocation, circuit edition[LB00]. Mapping a logic description to a particular technology is achieved using the algorithms packed into SIS[Sa92], or hierarchical and parallel synthesis from Lemarchand's PPart[Lem99]. Specific *atomic* resources can be merged with logic, and the framework is extensible.
2. **High level logic compiler.** This compiler produces circuits associated to high level functionalities on a characterization of the above model. Object oriented programming is not restricted to a particular set of operators or types, and so we provide the capability to produce primitives for arbitrary arithmetics or symbolic computing.
   At an intermediate level, the compiler handles a graph of lookup-tables carrying high level values (objects). Then this graph is translated into a logic graph that will be mapped on hardware resources. The translator makes use of values produced in the high level environment that allows to implement a lot of classic optimizations without attaching semantics to operations at the language level.
3. **System and architecture modeling.** The computation architecture in its static or dynamic aspects is described in this framework. For instance,

these are generic regular architectures with their associated tools, processes, platform management and system activity.
The compiler can make use of logic generation to produce configurations, bind them to registers or memories, and produce a configured application. The ability to control placing and routing given by the first part, and synthesis from the second part, allow to build complex networks of fine or medium grain elements.

The paper focuses on the logic compiler. Historically, this work has taken ideas from the Lin, Whitcomb and Newton symbolic translation to logic as described in [LWN91]. Relation with the object oriented environment has been described in [LP96] with limited synthesis capabilities that are removed in current work. System modeling and program synthesis has been demonstrated on the case study of a smart sensor camera[FLLP99] based on the same specification syntax as the one used in the current work.

The paper describes the general principles used for specification and logic production, then it gives more details on the transformations that are achieved. An illustration is given with the example of a coder/decoder family for RAID systems with quantitative results.

2 A framework for logic synthesis

2.1 Architecture modeling

Reconfigurable architectures can mix different grain of hardware resources: logic elements, operators, communication lines, buses, switches, memories, processors...

Most FPGAs provide logic functions using small lookup memories (LUT) addressed by a set of signals. As seen from the logic synthesis tools, an n-bit wide LUT is the most general way to produce any logic function of n boolean variables. There are known algorithms and tools for partitioning large logic tables or networks to target a particular LUT-based architecture.

LUTs are effectively interconnected during the configuration phases to form logic. This is achieved using various configurable devices such as programmable interconnect points, switches, or shared lines. Some commercial architectures also group several LUTs and registers into cells called configurable logic block (CLB).

Our model for the organization of these architectures is a hierarchy of geometric patterns of hardware resources. The model is addressed via a specific grammar[LB00] allowing the description of concrete architectures. Given this description, generic tools operate for technology mapping, placing and routing logic modules. See figures 6 and 7 for a view of the generic editor. Circuits such as operators or computing networks are described by programs realizing the geometric assembly of such modules and their connection.

Using this framework, few days of work are sufficient to bring up the set of tools on a new architecture, with the possibility to port application components.

On a concrete platform, it is then necessary to build the bit-stream generation software by rewriting the configuration descriptions to the basic tools. Two practical examples are the xc6200 that has a public architecture and has been addressed directly, and the Virtex addressed through the JBits API. We are also working on industrial prototype architecture.

2.2 Programming considerations

Applications for fine grain reconfigurable architectures can be specialized without compromise, and they should be optimized in terms of space and performance. In our views, there is an abusive advantage given to the local performance of standard arithmetic units in the synthesis tools and also in the specification languages.

A first consequence of this advantage is the restricted range of basic types coming from the capabilities of ALU/FPUs or memory address mechanisms. Control structures strictly oriented toward sequentiality are another aspect that can be criticized. As an example, programming for multimedia processor accelerators remains procedural despite all the experience available from the domain of data parallel languages. Hardware description languages have rich descriptive capabilities, however the necessity to use libraries led the language designers to restrict their primitives to a level similar to C.

Our aim is to produce a more flexible specification level with direct and efficient coupling to logic. This implies allowing easy creation of specific arithmetics representing the algorithm needs, letting the compilers automatically tune data width, and modeling computations based on well understood object classes.

To reach this goal, specifications with symbolic and functional characteristics are used, jointly with separate definition of data on which the program will operate. Data are objects that have binary representation.

Sequential computations can be structured in various ways by splitting programs on register transfers, either explicitly in the case of an architecture description, or implicitly during the compilation. In this case, high level variables are used to retain a state with known initial values, the compiler retrieving progressively the other states by enumeration [LP96]. Figure 1 shows a diagram where registers are provided to hold state values associated to high level variables that could be instance variables in an object.

In this paper we will consider the case of methods without side effect, operating on a set of objects. For sake of simplicity we will rename these methods '*functions*', and the set of objects, '*values*'. Interaction with external variables is not discussed in this paper. The input language is Smalltalk-80, variant Visualwoks, also used to build the tools and to describe the application architectures.

2.3 Execution model

The execution model targeted by the compiler is currently a high level replication of LUT-based FPGAs. We define a '*program*' as a function that needs

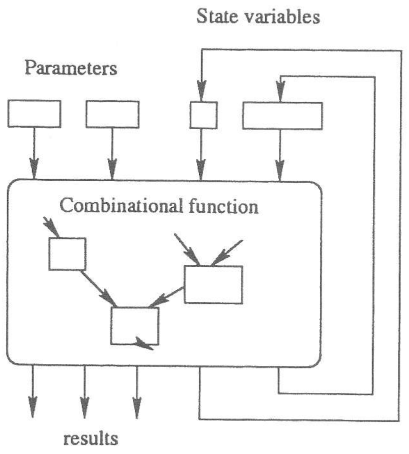


Fig. 1. State machines can be obtained by methods operating on private variables having known initial values.

to be executed on a set of input values. Thus the notion of *program* groups at once the algorithm and the data description. Our *program* can be embedded in higher level computations of various kind, implying variables or memories. Data descriptions are inferred from these levels. The resulting circuit is highly dependent from the data it is intended to process.

An execution is the traversal of a hierarchical network of lookup tables in which values are forwarded. A value change in the input of a table implies a possible change in its output that in turn induces other changes downstream. These networks reflect the effective function structure at the procedure call grain and they reflect an algorithmic meaning. Among the different possibilities offered for practical execution, there are cascaded hash table accesses, and use of general purpose arithmetic units where they are detected to fit.

Translation to FPGAs needs binary representation for objects. This is achieved in two ways, by using a specific encoding known to be efficient, or by exchanging object *values* appearing in the input and output for *indexes* in the enumeration of values. Figure 2 shows fan-in and fan-out cases with the aggregation of indexes in the input, and the *next index* selection from the table. Basically the low level representation of a node is a PLA having in its input the Cartesian product of the indexes, and in its output the aggregation of indexes for downstream.

There are some important results or observations from this exchange:

1. data paths inside the network do not depend anymore on data width but on *the number of different values present on the edges.*
2. depending on the interfacing requirements, it will be needed to insert nodes in the input and output of the network to handle the exchanges between values and indexes.

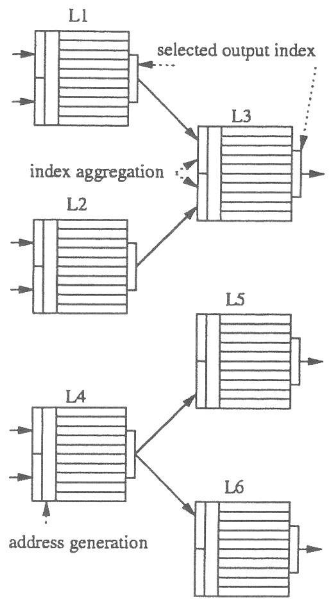


Fig. 2. Fan-in: index are aggregated to form an address in the table. Fan-out: the same index is presented to each table downstream.

3. logic synthesis tool capabilities are limited to medium grain problems. To allow code production for FPGAs, algorithms must *decrease the number of values* down to nodes that can be easily handled by the logic generation layer. Today, this grain is similar to the 8-bit microprocessor grain.
4. *decreasing the number of values* is the natural way in which functions operates, since the size of a Cartesian product on a function input values is the maximum number of values produced in the output. The number of values carried by edges is decreasing either in the hierarchy structure or in a graph flow. There is no possible divergence and the efficiency of an algorithm can be stated to be its ability to quickly decrease the data amplitude on which logic complexity depend.

2.4 Type system

Language types appear to the programmers as annotations for checking code consistency and binding to architecture resources. The type system we are using does not restrict programming to this kind of binding. It is only intended to specify any possible set of values appearing in the program input or inside the computation network. In the object environment, it is supported by a set of classes supporting operations.

Implicit or explicit collections of values are denoted by intervals or sets. *Class-based types* are associated either to classes having a finite number of instances (booleans, bytes, small integers), or to user defined new functionalities, including arithmetics. *Unions* are resulting from operations on the two previous types.

2.5 A short illustration

To illustrate the programming interface, let us comment the case of a multiplier operating on small floating point numbers. The numbers are represented as a sign, an exponent and a fractional part. The multiplication is described in a class supporting secondary methods for adding exponents, multiplying fractional parts, and normalizing the result.

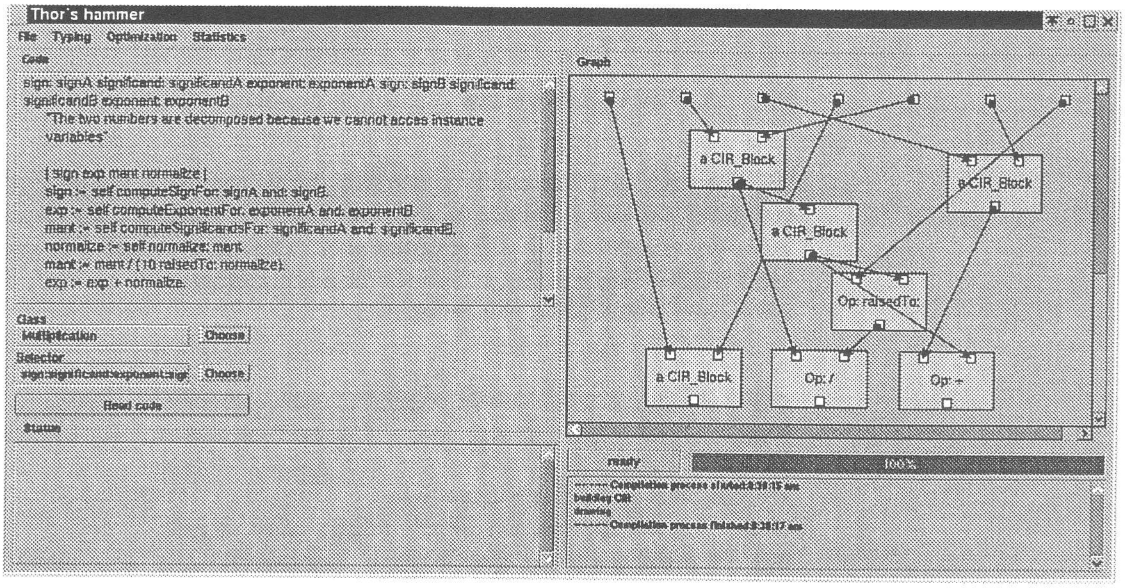
```
sign: signA significand: significandA exponent: exponentA
sign: signB significand: significandB exponent: exponentB

| sign exp mant normalize |
sign := self computeSignFor: signA and: signB.
exp := self computeExponentFor: exponentA and: exponentB.
mant := self computeSignificandsFor: significandA and: significandB.
normalize := self normalize: mant.
exp := exp + normalize.
mant := mant / (10 raisedTo: normalize).
```

Depending on the data amplitude in each node inputs, the compiler will make the choice to develop hierarchically or not. In the first case the node will directly have its table computed. In the second case a new graph is produced which is accessible from the node.

The code presented is executable in the high level environment. To be translated into logic, it is necessary to provide a characterization of the objects presented the various fields as boolean and intervals tailoring the arithmetic. In the case of parameters of small amplitude, in the order of 8bits, the compiler succeeds to produce the necessary logic locally optimized. For lower amplitudes synthesis could have been achieved directly, and a post-optimization effectively succeeds to improve the logic in an order of 25%. To fix an upper limit, a global post-optimization by standard SIS algorithms achieved on a 10 bits floating multiplier was achieved in 6 hours. The circuit was synthesized for a LUT-4 architecture, and the post-optimization decreased its size from 180 to 140 cells.

For higher amplitudes, it will be needed to split the fractional part and to handle it in an array.

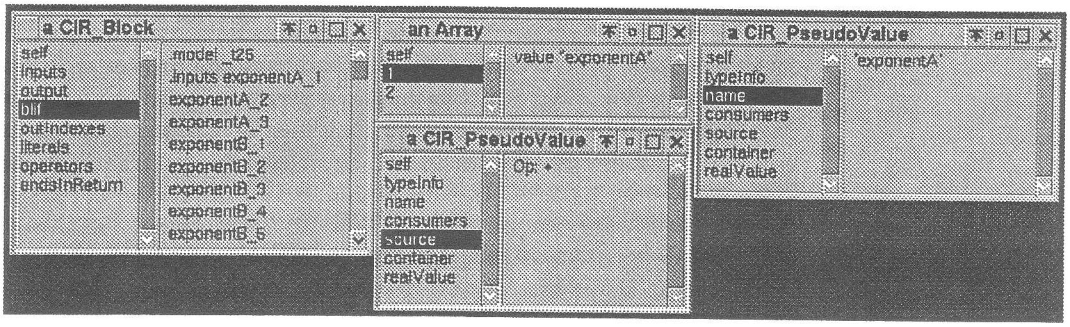
It can be noticed that these operators implement general operations on custom arithmetic. By restricting the operands to one, or a small set of values, the logic complexity can decrease dramatically at the point where the operation is simply a change in binary representation.



```
sign: signA significand: significandA exponent: exponentA sign: signB significand:
significandB exponent: exponentB
    "The two numbers are decomposed because we cannot acces instance
    variables"

    | sign exp mant normalize |
    sign := self computeSignFor: signA and: signB.
    exp := self computeExponentFor: exponentA and: exponentB.
    mant := self computeSignificandsFor: significandA and: significandB.
    normalize := self normalize: mant.
    mant := mant / (10 raisedTo: normalize).
    exp := exp + normalize.
```

```
------ Compilation process started 8:38:15 am
building CIR
drawing
------ Compilation process finished 8:38:17 am
```

Fig. 3. Development interface showing the high level code for floating point multiplier and the corresponding execution graph. The 3 nodes at the bottom of the figure output (left to right) the sign, the exponent and the fractional part. Boxes in the graph can be inspected to check the types carried on edges, as shown figure 4



```
.model _t25
.inputs exponentA_1
exponentA_2
exponentA_3
exponentB_1
exponentB_2
exponentB_3
exponentB_4
exponentB_5
```

Fig. 4. Inspectors on nodes of figure 3 graph. At the left, internal aspects of a node with inputs, output, and generated logic presented in the right pane in BLIF format. In the middle the inputs is an array of 2 parameters. The 2 other windows show value characterizations for the output and an input.

3 Compiler flow

3.1 Flat expressions

In a first stage, let us consider a program where the number of values appearing in the input of each function call is compatible with an efficient logic synthesis for a LUT-n FPGA architecture. As each node can be directly synthesized, we have a *flat* expression in opposition with *hierarchical* expressions that will need additional compilation contexts for some of the function calls.

As a Smalltalk development environment is used, there were an obvious interest to use this language syntax for '*programs*' targeting FPGAs. Immediate benefits are the reuse of the standard compiler front-end and use of the existing classes.

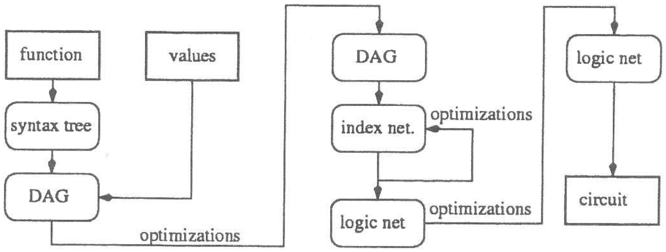


Fig. 5. Compiler flow

1. **Building the value network**
   The first compilation stage consists in building an acyclic flat graph which nodes are lookup tables based on objects and which edges allow to pass values downstream.
   As stated, the syntax tree is produced by the standard compiler. The directed acyclic graph (DAG) is built by analyzing the syntax tree and variable use. Local variable references are eliminated. At this stage nodes are still holding function calls receiving edges from the function parameter list, or other nodes.
   To replace these nodes by lookup tables, the values are propagated progressively from the function parameter list. A graph traversal is achieved, building a table for each node having defined inputs.
   During this transformation care must be taken of dependencies in variable used in fan-out to fan-in subnets. As an example, the composition $h(f(x,y),g(x,z))$ has a smaller output than $h(f(x,y),g(t,z))$ because of the dependency on variable x. A number of inputs in the fan-in node h and upstream are not useful and can be deleted by constraining the Cartesian products in f and g tables[1]. Lot of conditional computations fall in this case.

[1] This mechanism is very important but is not yet implemented at this date

2. **High level optimization and building the index network**
   After this first stage we have a situation similar to a compiler having a language semantic knowledge because the tables have inferred stronger properties from the message executions. It is time to apply high level optimizations such as elimination of constant nodes and dead code or subexpression factorization. This imply backward and forward processing on the DAG.
   The next transformation is the translation of the DAG by deducing index based tables from associations of value tables. This is achieved by generating index for values. Care must be taken of class based types to preserve their special encoding.
3. **LUT based optimizations and architecture mapping**
   Index path optimizations involves the detection of subnets with particular topologies. As an example, linear cascade of tables can be collapsed in a single table. For logic translation, each index-based table is given to logic synthesis tools to produce an equivalent binary description. At this stage we must also take into account the size of LUT memories in the target architecture. The result is a hierarchical logic description which is a binary equivalent for the high level program.
   The last stage is to place and route the logic graph using the generic tools in the framework, producing a hardware module for further system handling and binding.

3.2 Hierarchical aspects

In section 3.1 we supposed that the program can be directly synthesized at each function call. We are now considering the more general case where calls must be developed to reach this condition.

The logic needed to implement a particular function call depends on the expressed algorithm, the number of parameters, the number of possible values for parameters and the original encoding of values in the higher level environment. A valuable property of an algorithm is its ability to quickly decrease the number of values present on graph edges. This gradual decrease comes from function calls that are processed in the same way as their root function, for every node showing an excessive complexity related to synthesis.

When the compiler reaches a condition where logic tools will be inefficient, it creates a new compilation context and process recursively the call. The context will return a structured logic description that will be installed as part of the current level production.

The technical form of a logic description associated to a compiled program is a hierarchical BLIF description that can be partially flattened for further logic optimization, and partially placed under control of a floor planner. In this case each developed function call has its corresponding circuit component assembled in the global hierarchy.

A more speculative compiler built-in function is type partitioning. When a data set appears to be too much large, the compiler can divide the type in order to reach a grain suitable with synthesis. Automatic type division by the compiler should be considered only as a quick approximation, since the *function* algorithms are normally written to manage synthesis complexity at high level.

A similar situation is the knowledge of a 'best encoding' for values. As an example, the order of elements in a Galois field has an influence on the logic complexity of basic operators. If these operations are dominant in the code, type-based rules must be managed by the compiler to prevent new type generation in node outputs.

4 A RAID error correction case study

The procedure for flat expressions is illustrated by the example of a RAID system correction system. In RAID system redundancy for error correction is kept using Reed-Solomon (RS) coding over Galois Fields. Here we have made the choice of a field with 2^4 elements. We will concentrate on the implementation of the encoder/decoder parts. We will talk of $n : m$ RS indicating a RS schema where m checksum disks are used as redundancy for n disks of data.

Basically, the encoder part will take the n streams to be stored in the data disks to generate the m streams to be stored in the redundant disks. This makes for a unique reconfiguration for a given $n : m$ schema.

On the other hand, the decoder part will take the streams stored on disks and return the original data streams. When all stored streams are available, the decoders simply return the data from the data disks. However, if one (or up to m) disks fails, the original data may need to be reconstructed from the checksums.

The use of FPGA for encoding/decoding seems appropriate mainly for two reasons:

- There may be little or no performance loss due to error correction. The cost is paid only when a disk failure happens (transition from disk working to disk non-working).
- It will provide the system with added flexibility. The ability to mutate the circuits will allow the same hardware to be used for different failure schemas.

We have based our case study in the encoding/decoding in a word-by-word basis. This means that, as we are talking of Galois Fields 2^4, we obtain circuits that take data from the streams in groups of 4 bits. As the operations for different words are independent, it is possible to replicate the circuits to work on multiple data words simultaneously (at the expense of logic space) to meet desired performance.

4.1 Reed-Solomon introduction

Reed-Solomon (RS) coding allows correction of up to m errors using m checksum words. For a system with n data words allowing error recovery for m failures, a total of $n + m$ words should be stored.

The basic idea of RS coding is to build a system with $n + m$ rows and n columns. All rows are built to be independent. Recovery from up to m errors is possible as we could always take the available words and build a system that is solvable. Solving that system by any technique (like Gaussian Elimination) will provide the original data words.

Encoding: at this point we need to build m additional independent equations. To achieve that, we will use the Vandermonde matrix and compute its associated independent terms that will serve as checksums. This can be done by performing the following operation (where $d_1 \ldots d_n$ are the data words to encode and $c_1 \ldots c_m$ its associated checksums):

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}. \quad (1)$$

Decoding: when retrieving data we know that the following equation must apply:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}. \quad (2)$$

So in case of an error on a word of data, we can compute its value by solving a system involving n rows of the equation. This will be possible as long as we have n valid values in the independent term vector; that is, there are less than m errors.

Arithmetic over Galois Fields is used as the algebra needed to solve the system, as is closed over a field of finite size.

For a more detailed description of RS coding using arithmetic over Galois Fields, the reader may refer to Plank's tutorial[Pla99]. Other interesting bibliography comes from C. Paar et al., for example [PR97] provides information on GF operator complexity and implementation on FPGAs.

4.2 Encoder/decoder specification

Our objective is to obtain configurations for the encoding/decoding using RS over Galois Fields. We are targeting a system that has a reconfigurable part to do both, encoding and decoding. We will need a configuration for the encoder and several configurations for decoding, one for each working condition (set of words missing). This can be applicable to RAID systems, where the working condition (disk failure) can be considered rare, and the cost of reconfiguration in such a case will have little impact.

The specification has been developed in three steps:

Reed Solomon specification. The specification generates the equation 2. This is specified as a Smalltalk class that has methods for encoding/decoding data. The class has been build in order to allow the number of data and checksum words at instance creation, so it can be used for any encoding size. The specification has been tested by exhaustively performing error correction using conventional arithmetic.

Development of Galois Field arithmetic class. A Galois Field 2^4 for Smalltalk has been developed (GF16). The operations implemented are those needed in our problem, following the guidelines shown in [Pla99]. Using Smalltalk's powerful polymorphic nature, we can apply the Reed Solomon class using the new arithmetic. We have performed an exhaustive test of the RS coding using the Galois Field arithmetic in order to test correctness.

Extraction of error correction expressions. Building an arithmetic-like class that records the operations performed, we can build the expressions for the encoding of checksums, as well as for the decoders in a given working condition. Those expressions can be packed into method code that will be compiled to build the configurations.

As a note, the above specification took a few hours to be done, with no initial experience on Reed-Solomon coding.

4.3 Expression compilation

From the specification, we can take the expressions for the encoders/decoders and compile them to logic. The steps below show the most important effect of the implementation as handled in our framework.

Type inference. After building the DAG, all edges are typed. In this case, the inputs are data and checksum words, all of type GF16.

Constant folding. Due to the generated nature of our expressions, there are plenty of operations over constants. Those are all removed in this steps. Also operations wielding a constant result (like multiplication by 0) are removed.

Dead code removal. As a result of removing multiplication by 0 operations, and due to the nature of automatically generated expressions, it is possible that there are expressions whose result is never used, so they are removed.

Code factorization. That is, common sub-expression elimination.

Operator LUTification. This step is the first one towards architecture binding. It transforms the symbolic operations into *look up tables* suitable for logic synthesis.

No-op removal. Unary operators whose output is equivalent to its input are removed.

Operator fusion. Unary operators are removed by fusing them with its producer/consumer operators. We assume that this will provide a better implementation.

Circuit production. Several circuits have been produced to collect practical information of the results.

4.4 Encoders/decoders statistics

Tables 1 and 2 display respectively statistics for the 3 necessary encoders and all the possible decoders for disk failures. The tables has columns showing the decrease in the number of GF16 operators, average number of inputs per operator, and the critical path in the network of operators. The meaning of rows is the observed value after each optimization operation as described in section 4.3.

Correctness has been checked at the logic level by selecting random inputs and verifying the output *after logic synthesis* using SIS *simulate* command.

Table 1. statistics for encoders-RS4:3.

Compiler operation	operators	average input	critical path
type inference	12	2	6
constant folding	8	2	5
dead-code removal	8	2	5
code factorization	8	2	5
operator to LUT	8	1.375	5
no-op removal	5	1.67	3.67
operator fusion	3	2	3

Table 2. statistics for decoders-RS4:3.

Compiler operation	operators	average input	critical path
type inference	85.08	2	11.24
constant folding	11.68	2	7.65
dead-code removal	11.68	2	7.65
code factorization	10.41	2	7.65
operator to LUTs	10.41	1.43	7.65
no-op removal	7.42	1.65	5.75
operator fusion	4.5	2	3.625

Table 3. Compiler result for disk 1 decoder, with disks 1 and 2 broken

Compiler operation	operators	average input	critical path
type inference	90	2	15
constant folding	30	2	12
dead-code removal	30	2	12
code factorization	30	2	12
LUTification	30	1.47	12
no-op removal	21	1.67	9
operator fusion	14	2	8

Table 4. Compiler result for disk 2 encoder

Compiler operation	operators	average input	critical path
type inference	24	2	10
constant folding	16	2	9
dead-code removal	16	2	9
code factorization	16	2	9
LUTification	16	1.44	9
no-op removal	14	1.5	8
operator fusion	7	2	7

4.5 Specific 8:2 case with circuit generation

This time, the case of a RAID system with 8 data disks and 2 redundant disks is considered. The compiler result is given for the encoder on error correction disk 2 (Table 4), and then for the decoder on first data disk, with data disk 1 and 2 in failure (Table 3).

These circuits has been optimized and mapped to 2 different architectures having 2-LUT and 4-LUT cells. The table 5 shows the compared characteristics of the encoder and decoder on these architectures. Each one has a routing channel of size 8 inside the cell patterns, providing a first run success. Some parameters are extracted that can be used at a higher level, as an example for system management of the reconfigurable logic resources, or for making choices in the compiler generation code strategy. Notice that at the end of optimization on LUT, it is easy to generate processor code and table contents equivalent to the network of reconfigurable logic cells.

The table has two parts for post-assembly optimized logic and simple structured assembly as it is used for floor planning. The presented characteristics are:

1. total circuit area in number of cell patterns,
2. gates used in this area,
3. number of inputs for the circuit,
4. effective number of cells used to implement logic,

5. the average of used inputs in module cells including the border
6. the same measure for cells in (4)

Notice that (3)+(4)=(2), with (4) being low. The circuit is I/O dominated. Gates used in (3) disappear when the module is connected to other architecture element.

Routing cost (7) is an estimation on the number of resources allocated for connections. Critical path (8) is the maximum number of cells and other resources allocated in the circuit between an input and an output, with unitary costs. CPU time (9) provides an idea of the delay to place an route the circuit on a PC/750Mhz with the Visualworks environment running on Linux. Figures 6 and 7 are views on these decoders as generated by the tools.

(10) is the maximum area occupied by the assembly of elementary modules without post-assembly optimization, and without the use of the floor planner. (11) is the maximum number of cells used in this area, and (12) is the number of cells used to implement logic in the area. (12) is similar to (4). A good measure of the post-assembly optimization is the respective 46% and 27% logic decreases in the cases of the decoder and encoder. The use of the floor planner will bring (10) and (11) closer to (1) and (2).

Table 5. Results from place and route on 2 architectures

	Encoder		Decoder	
	LUT 2	LUT4	LUT2	LUT4
Area (1)	90	56	121	72
Cells Used (2)	85	53	119	71
Input cells (3)	32	32	40	40
Internal cells (4)	53	21	79	31
Input average (5)	1.62	2.04	1.67	2.23
Gates Input average (6)	2.0	3.62	2.0	3.81
Routing Cost (7)	1095	640	1839	1049
Critical Path (8)	18	14	19	15
CPU Time (9)	43.14	20.34	98.70	34.89
Max struct. area (10)	128	88	208	176
Cells used (11)	109	85	181	170
Internal cells (12)	53	29	79	58

5 Conclusion

The section 1 has presented our project in three parts. Reconfigurable architectures modeling tools are operational, with a practical implementation on xc6200 and active work to address the Virtex. Another development is in progress for a low power FPGA prototype architecture. Using these tools, one can describe

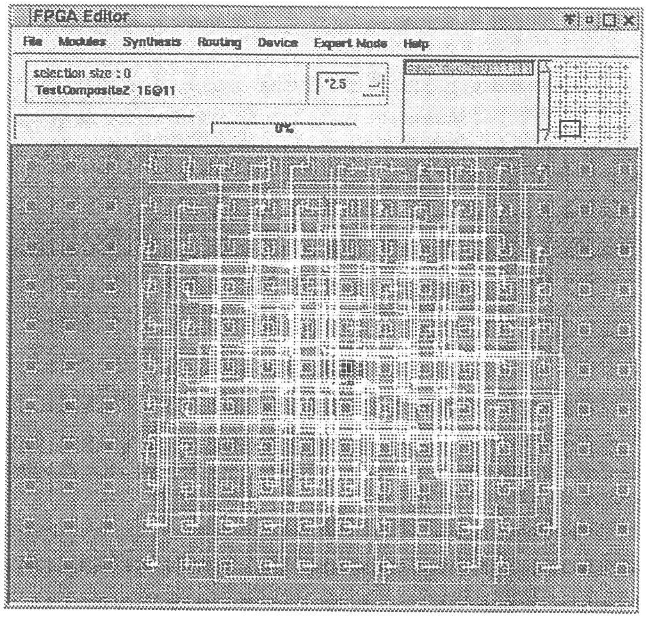


Fig. 6. A decoder for an 8 data and 2 redundancy disks problem placed and routed on a Lut-2 architecture

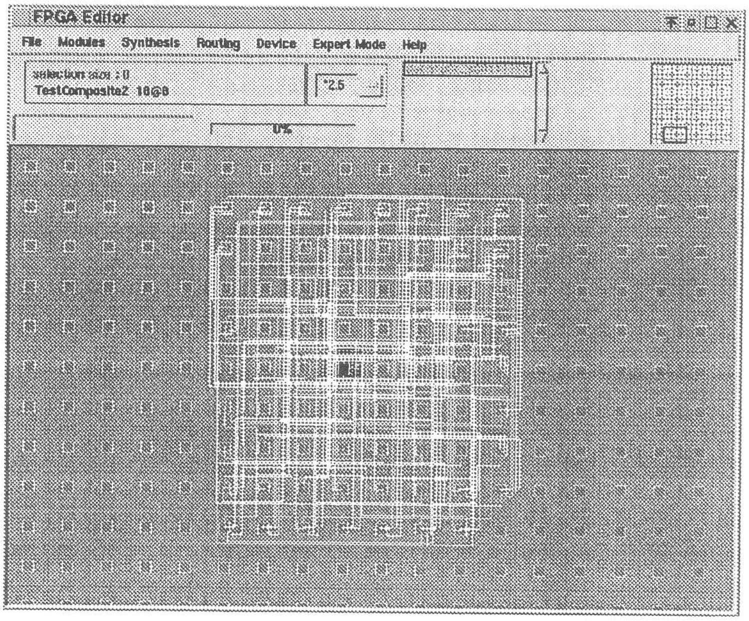


Fig. 7. The same decoder on a Lut-4 architecture

regular circuits for operators or processing networks by replication and channel routing.

The compiler described in this paper is a work in progress. With the exception of optimization for variable dependencies, it is now possible to produce an optimized hierarchical logic description suitable for technology mapping, then place and route. This compiler handles optimization mostly at high level, removing a considerable load on logic mapping algorithms. The execution model can be understood as a specific lookup memory unit linking symbolically input stimuli to outputs. The strength in optimization comes from the fact that the knowledge of values being processed allows to simplify computations either at high level or at logic level.

The most important point is that this method gives the possibility to create specific logic based on concise behavioral algorithm expression that is reusable in a variety of situations on different kind of data. Binding nodes to memories or arithmetic operators is feasible based on the architectural model and types propagated inside the computation graph.

We find in the object oriented approach very promising results either for architecture management and high level synthesis which must be considered as a productivity tool in the context of systems on chips.

References

[DC90] E. Debaere and J.M.V. Campenhout. *Interpretation and instruction path coprocessing.* MIT Press, 1990.

[FLLP99] G. Fabregat, G. Leon, O. Le Berre, and B. Pottier. Embedded system modeling and synthesis in oo environments. a smart-sensor case study. In G. Gao and K. Palem, editors, *CASES'99*, Oct 1999.

[LB00] L.Lagadec and B.Pottier. Object oriented meta-tools for reconfigurable architectures. In *SPIE, Reconfigurable technology II*, volume 4212, November 2000.

[Lem99] L. Lemarchand. Parallel performance directed technology mapping for fpga. In *Proceedings of IEEE Southwest Symposium on Mixed-Signal Design, Tucson, USA*, pages 189–194, 1999.

[LP96] José-Luis Llopis and Bernard Pottier. Smalltalk blocks revisited, a logic generator for fpgas. In J-M. Arnold and K. Pocek, editors, *FCCM'96*, Napa, CA, 1996. IEEE press.

[LWN91] B. Lin, S. Whitcomb, and A. Newton. Symbolic don't care and equivalence in high level synthesis. In (IFIP) P. Michel and G. Saucier, editors, *Logic and Architecture Synthesis.* Elsevier, 1991.

[Pla99] James Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. Technical Report UT-CS-96-332, Department of Computer Science, University of Tennessee, February 1999.

[PR97] C. Paar and M. Rosner. Comparison of arithmetic architectures for reed-solomon decoders in reconfigurable hardware. In *(FCCM'97)*, Napa, CA, 1997. IEEE press.

[Sa92] E.M. Sentovich and al. Sis: A system for sequential cirquit synthesis. Technical Report UCB/ERL M92/41, EECS, Berkeley, May 1992.

In Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece, July, 2002.

Goal-Driven Reconfiguration of Polymorphous Architectures

Sumit Lohani and Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park MD 20742, USA
{slohani, ssb}@eng.umd.edu

Abstract. Polymorphous computing architectures refer to computing platforms whose computation and communication structures can be changed over time. The objective of such platforms is to use the underlying reconfigurable components and attributes to adapt to dynamically changing constraints, objectives, and characteristics in the applications that execute on them. In this paper, we present a model for executing applications with non-deterministic execution times and time-varying performance requirements on a polymorphous architecture. We analyze the complexity of various issues related to the model, and identify some broadly-applicable conditions under which the complexity of these issues can be reduced. We develop a heuristic framework for guiding the run-time configuration adaptation process, and show through simulation experiments that this approach can efficiently handle both dynamics in performance requirements and in task execution times.

1. Introduction

There have been many advancements in recent years on architectures for reconfigurable processing engines (e.g, see [7][12][13]). With the increasing degree of reconfigurability in processing architectures, it is useful to view embedded multiprocessor systems as polymorphous computing architectures (PCAs) in which configurable attributes of the architecture and software are adapted in response to dynamically changing needs. Such attributes may include items such as inter-processor message routing, caching policies, scheduling policies, processor voltages, resource allocation to computing units, and synchronization protocols. A PCA can be a particularly useful platform for developing a computing system where applications and performance requirements change at run-time as one can adaptively configure the PCA to suit the dynamic constraints and objectives.

This paper takes a step towards bridging techniques for scheduling and system synthesis with reconfigurable processing platforms and the dynamically-changing application requirements that drive these platforms. We first formulate the problem of executing application dataflow graphs on a polymorphous computing architecture such that specified performance requirements are satisfied, where the requirements may vary over time and the application may have tasks with non-deterministic execution times (e.g., due to data dependencies or unpredictable events such as cache misses and interrupts). We analyze key properties of this problem and the complexity of some relevant sub-problems. We then develop a flexible heuristic framework for guiding the run-time configuration adaptation process, and show through simulation experiments that this approach can efficiently handle both dynamics in performance requirements and dynamics in task execution time behavior.

In the application model addressed in this paper, computational tasks (*actors*), which are represented by dataflow graph vertices, in the application are allowed to have stochastic execution times with static distributions or distributions that may vary slowly over time. The computing unit is a reconfigurable multiprocessor architecture, and the objective is to find a mapping of the actors in the application onto the processors in the multiprocessor and the configuration that the architecture should assume such that performance-related constraints (e.g., constraints on power, resource usage or throughput) are satisfied and objectives (e.g., maximizing throughput or minimizing latency) are optimized effectively. Furthermore, the constraints and objectives may vary over time, and thus, overall solution quality can be viewed in terms of how efficiently reconfiguration of the architecture tracks changes in the application's requirements. Henceforth, we will refer to this problem as the polymorphous computing architecture mapping (PCA mapping) problem. As can be seen, the PCA mapping problem is quite general in nature and even very restricted special cases can be proved to be NP-complete.

The approach suggested in this paper is correspondingly general and can handle diverse applications and performance requirements. All the reported experiments were performed on an abstraction of the Raw architecture [12] that incorporates salient features of the architecture such as the programmability of interconnects between processors. For experiments, the self-timed execution of applications on this abstracted Raw architecture was simulated using the inter-processor communication (IPC) graph model [12].

The emphasis in this paper is on coordination of the on line configuration management process for reconfigurable networks of processors, rather than the development of specialized configuration optimization techniques (such as fixed-objective scheduling and allocation), which are already in abundance in the literature (e.g., see [11] for a survey). Our work is complementary to such existing efforts and also to work on multiprocessor system synthesis [1][2], which can be used to derive the store of pre-computed configurations that is input to the techniques developed in this paper.

2. Problem formulation

A set of relevant metrics, such as latency, throughput, average power, peak power, and number of resources, is denoted by M. If a certain metric appears as a constraint with a value to be satisfied when the application executes, then this metric is referred to as a *constraint metric* and the value as a *constraint value* for that particular metric. A constraint value belongs to the set of real numbers. A pair of constraint metric and constraint value is called a *constrain pair.* A sequence of constraint pairs in turn is referred to as a *constraint vector,* and is denoted by

$$V = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K)], \tag{1}$$

where $m_1, m_2, \ldots, m_K$ represent any K metrics in M, and $c_1, c_2, \ldots, c_K$ represent the corresponding constraint values, for $K \in \{0, N\}$, where N is the number of all constraint pairs. This (possibly empty) sequence of constraint pairs in a constraint vector is prioritized such that (m_i, c_i) is a higher priority constraint pair than a constraint pair (m_j, c_j) if $i < j$, for $i, j \in \{1, 2, \ldots, K\}$ in a constraint vector $V = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K)]$. A metric m_R that is to be optimized after all constraints have been satisfied is called a *residual objective*. A *goal* g is an ordered pair (V, m_R), where V is a constraint vector and m_R is a residual objective. If there is no residual objective, then the goal is composed of only a constraint vector and can be represented by $(V, \perp)$. Here, the symbol $\perp$ represents the absence of a residual objective. Also, without loss of generality, the metrics are such that the associated optimization problems are to *minimize* the metric (i.e., a lower value of a metric is always better than a higher value). Metrics for which higher values are more desirable must thus be transformed into corresponding metrics for which lower values are better. For example, in iterative applications, the throughput (average rate of completion of application iterations) can be re-cast as the *average iteration period*, which is the reciprocal of the throughput.

Example 1: Consider a set of relevant metrics $M = \{L, P, T\}$, where L is the latency, P is the average power consumption, and T is the iteration period. Consider the goal $g = [(L, 50), (P, 100), (L, 40), (P, 70), T]$. In g, the constraint pair $(L, 50)$ has higher priority than the constraint pair $(P, 100)$, which in turn has higher priority than the constraint pair $(L, 40)$. The metric T is the residual objective.

This definition of reconfiguration goals as prioritized lists with optional residual objectives leads to a view of dynamic reconfiguration as a sequence of one-dimensional optimization problems. This simplification is useful because run-time adaptation techniques must be of relatively low complexity, and thus, one-dimensional optimization is a better match. Additionally, it allows us to leverage existing libraries of single-dimensional synthesis techniques, which are more abundant than multi-dimensional techniques. Third, it provides an intuitive and unambiguous format for designers to prioritize multidimensional application requirements. Note, however, that this formulation applies only to run-time reconfiguration, and multi-dimensional optimization techniques, such as SPEA-based methods [14], can be used off-line in arbitrary ways to compute caches of pre-computed configurations. Use of such caches will be discussed further in Section 3.2-5.

For example, in Example 1, we initially have an unconstrained latency optimization problem (since the first constraint involves latency). As we adapt the system configuration with techniques that address this problem, we will in general improve the latency. Once the latency improves to 50 time units, the current constraint is satisfied, and we switch to a power-optimization problem subject to a constraint of $L = 50$. The optimization process may continue in this manner until the last constraint is satisfied (in this case, $P = 70$), at which point run-time adaptation stops (if there is no residual objective) or reaches a terminal mode of optimizing the residual objective subject to all constraints in the constraint vector. This mode then continues until the system shuts down or the application's goal changes.

Mapping an application to a multiprocessor architecture includes defining a task-to-processor mapping along with defining the configuration of the reconfigurable architecture. In this paper, the scope of the word "configuration" is expanded to include also the mapping of the application onto the reconfigurable architecture.

Therefore, a *configuration* consists of two components 1) task-to-processor mapping and 2) configuration of the architecture. Henceforth, the word "configuration" is used in the above sense, unless stated otherwise. A given application, goal, and resource set define an *instance* of the PCA mapping problem. Input to the model is an instance that may change with time. We define the *design space* as the set of all feasible combinations of an instance and a configuration. The *solution space* for a feasible instance is the set of all feasible configurations for that instance. Latency, throughput, average power and peak power are some of the commonly encountered metrics. With many metrics of simultaneous relevance, the goal space is too vast to be fully explored before run-time, and run-time adaptation of configurations is generally advantageous.

Figure 1 illustrates a general model for solving the PCA mapping algorithm with a combination of off-line and on-line techniques. The main components of the model are the *off-line component,* the *configuration store* (*CS*), and the *on-line component*. The off-line component, whose objective is to pre-compute a set of efficient candidate mappings for various run-time scenarios, can be constructed using existing methods for scheduling, system synthesis, and multi-objective optimization. The focus of this paper is thus on the on-line refinement component and its interaction with the configuration store.

For a given instance, not every configuration is suitable as some configurations may violate constraints or may not adequately address residual objectives. As the goal changes for a given application, the system needs to derive a suitable adaptation of the run-time configuration. Optimally solving this problem is undecidable in many contexts. Also, reconfigurability of the architecture and the stochastic variance of execution times greatly complicates the solution space consisting of all possible configurations for the input of a goal and a given application. Since computing a suitable configuration is performed during the execution of an application, one can not apply exhaustive or relatively sophisticated search strategies as those techniques will take away excessive computational resources away from the application itself. To address this trade-off (thoroughness of dynamic optimization vs. resources drained from the application), our model of the PCA mapping problem also accounts for the time spent in computing efficient adaptations of mappings at run-time on the basis of feedback obtained from execution and identification of bottlenecks, and hence always tries to move towards an optimal solution. This is taken care of in the on-line refinement part of the model, which consists of low-complexity algorithms that find and refine configurations for a given instance. It also consists of feedback units shown by the "*Identify bottlenecks*" block in Figure 1 that takes feedback from the execution of the configurations and modifies the configurations so as to better suit the active goal. The *OnlineStats* unit in the on-line refinement part of the model stores short-term statistics that can be used by on-line algorithms.

A configuration store is used to store high-quality points in the design space that have been explored so that one can use them later as need be. The off-line refinement part of Figure 1 consists of high-complexity algorithms that yield better solutions. It is acceptable for them to be of high-complexity as they are used off-line, and do not compete for resources with the application. In Figure 1, the *STATS* unit stores statistics about the application (e.g., distributions of execution times for different actors, frequencies of occurrence of some particular regions of the goal space, etc.). Off-line algorithms use these statistics to explore the solution space for input instances.

As soon as the goal or application changes, an initial configuration is found using the on-line configuration management component in conjunction with the configuration store. On-line algorithms keep improving the configuration that is being executed, using the feedback from the execution. In the meantime, off-line algorithms may keep exploring areas of design space and merge the relevant information into the configuration store (for use in the selection of future initial configurations).

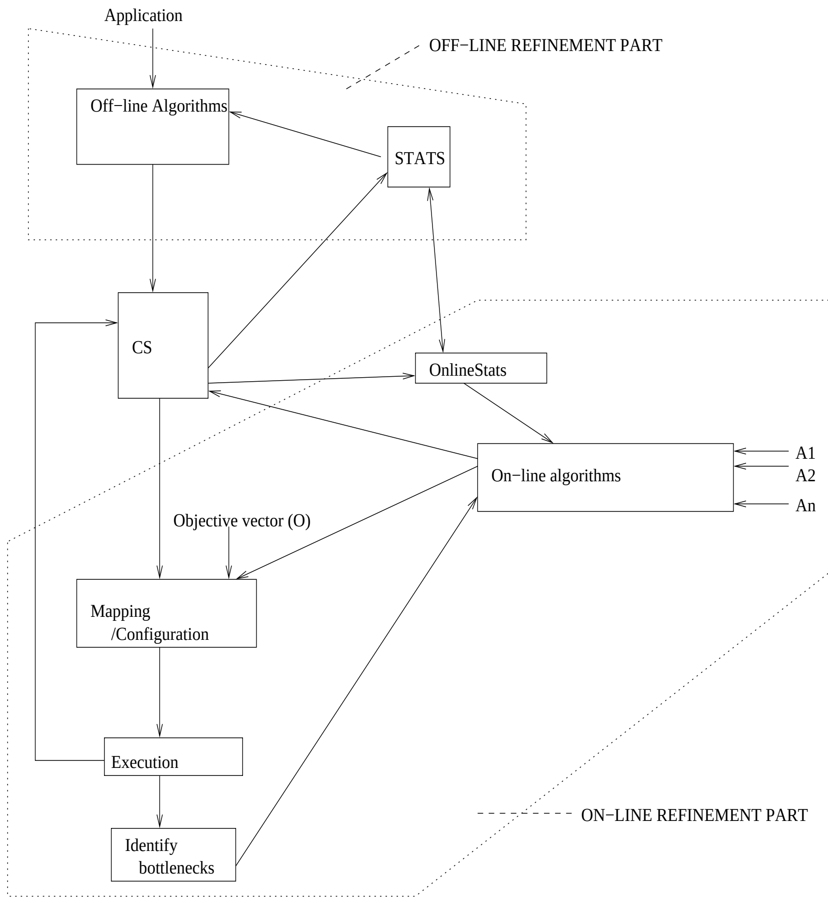


Fig. 1. An overview of the system-level reconfiguration framework studied in this paper.

3. Configuration management model

The overview of our PCA system synthesis model shows that it is very adaptive in nature and hence is suitable for applications with stochastic execution times and time-varying goals. This section develops further details of this model.

3.1 Evaluation of configurations and goals

It is useful to define some measure of how well a given configuration executes for a particular instance. This evaluation measure should allow unambiguous comparison between two configurations based on the current goal.

Suppose we are given a goal $g = [V, m_R]$, where

$$V = [(m_1, c_1), (m_2, c_2), \ldots, (m_n, c_n)] . \qquad (2)$$

We define the *quality* of a system configuration C with respect to goal g, denoted $Q_g(C)$ (or simply $Q(C)$ if g is understood) as the ordered pair $Q(C) = (k, v)$, where $k + 1$ is the index of first unsatisfied constraint in the constraint vector of g (i.e., the lowest-index constraint in g that is not satisfied by the configuration), and v is the value obtained for the metric m_{k+1}. If configuration C satisfies all n constraints in the constraint vector of g, then we say that C *satisfies* g, and in this case, $Q(C) = (n + 1, v_R)$, where v_R is the value obtained for the residual objective m_R if $m_R \neq \perp$ or $v_R = -\infty$ if $m_R = \perp$.

In summary, the quality of a configuration measures a configuration with respect to a given goal, and given a goal and two configurations C_1 and C_2 with qualities $Q(C_1) = (k_1, v_1)$ and $Q(C_2) = (k_2, v_2)$ for that instance, respectively, C_1 has higher quality than C_2 if

$$(k_1 > k_2) \text{ or } ((k_1 = k_2) \text{ and } (v_1 < v_2)) . \qquad (3)$$

3.2 Configuration store

A configuration store serves as a repository of alternative configurations. A configuration store can be divided into several sub-stores (sub-CSs), one for each relevant application. Each sub-CS has some configurations stored in it, one for a specific combination of goal and resource set. In the later part of this section, we assume that we are dealing with a fixed application and a fixed resource set, unless stated otherwise. This does not detract from the generality of the ideas developed later as they can be generalized to include various applications and resource sets using the hierarchical model of configuration store explained above.

Assuming a fixed application and resource set, selecting the goals whose corresponding configurations should be stored in the configuration store depends on various factors such as the size of the configuration store; the optimality of the stored configuration; computational resources drained from the application during execution by the on-line refinement algorithms; and the expected or observed frequency of specific goals.

3.3 Acceptability of configurations

Notions of *acceptability* and *cover* emerge naturally from this concept of configurations stores, and guide the construction and adaptation of the configuration store in our model. For example, one can envision the reconfiguration process as selecting an acceptable configuration, and gradually tightening the notion of acceptability to guide the on-line refinement process. The following definition makes these notions precise.

Definition 1: Given two goals g_1 and g_2, we say that g_1 is *acceptable* for g_2, denoted $g_1 \rightarrow g_2$, if a configuration that satisfies g_1 is an acceptable implementation for g_2. If $g_1 \rightarrow g_2$, we also say that g_1 *covers* g_2. Given a set Γ of goals and a specific goal g, the *space* of g over Γ (or simply, the *space* of g, if Γ is understood) is $\{g' \in \Gamma | g \rightarrow g'\}$. Thus, the space of a goal g is the set of goals that are acceptably implemented by any configuration that satisfies g. The space of a goal g is represented by $space(g)$.

The following result, proved and elaborated on in [9], shows that the acceptability of configurations is a particularly well-behaved relation if it is a partial order.

Theorem 1: If we have a finite set Γ of relevant goals, and the acceptability relation is a partial order, then there exists a unique, minimal set of goals $\{g_1, g_2, \ldots, g_n\}$ such that

$$\bigcup_{i=1}^{n} space(g_i) = \Gamma, \tag{4}$$

and this set of goals can be computed in polynomial time in $|\Gamma|$, the number of relevant goals.

Definition 2: *Dominance relation*: A point $p \varepsilon \Re^n$ dominates a point $q \varepsilon \Re^n$ if $p_i \le q_i$, for all $i = 1, \ldots, n$, where p_i and q_i denote ith components of p and q, respectively.

One can see that the dominance relation is a partial order [5]. We can have an acceptability relation between goals based on the dominance relation where a goal g_1 is acceptable for a goal g_2 if the constraint vector of the goal g_1 dominates the constraint vector of the goal g_2, and the residual objectives for both goals are same. The following example illustrates an acceptability relation that is not a partial order.

Example 2: Suppose that we have a single constraint metric, which is the average iteration period T of the system. Thus, the constraint associated with a goal g can be expressed as the desired average iteration period $T(g)$. Suppose that in a particular implementation context, the acceptability relation $g_1 \rightarrow g_2$ is defined by $T(g_1) - T(g_2) \le \Delta T$ for some positive real number ΔT. Thus, a configuration for g_1 can be worse than what is desired under g_2, and still be acceptable for g_2, as long as the deviation does not exceed the threshold ΔT. Suppose also that the goals g_1, g_2 and g_3 have desired average iteration period values of

$$T(g_1) = 5\,,\; T(g_2) = 5 - \frac{3\Delta T}{4} \text{ and } T(g_3) = 5 - \frac{3\Delta T}{2}. \tag{5}$$

One can then see that $g_1 \rightarrow g_2$ and $g_2 \rightarrow g_3$ but g_1 is not acceptable for g_3. Therefore, this acceptability relation is not transitive, and thus, is not a partial order.

An acceptability relation between goals based on the dominance relation or any other partial order leads to valuable properties such as that exposed by Theorem 1.

Also, the dominance relation is a natural candidate for an acceptability relation among goals, as a configuration corresponding to the dominating goal can be used in place of a configuration corresponding to the dominated goal without violating any constraints. This motivates our use of the dominance relation in managing configuration stores. One can observe that our approaches of defining a goal and the quality of a configuration are all consistent with acceptability based on the dominance relation.

4. On-line configuration management

In this section, we define an on-line configuration management framework called *CMF* that defines how to choose an initial configuration for a particular instance, and how the on-line adaptation for that configuration should proceed. We also formulate problems related to storage of configurations in the configuration store. These problems and our models to solve them provide fundamental analysis of the complexity of configuration management and provide feasible, low-complexity solutions to this problem.

A pseudocode outline of the CMF approach is shown in Figure 2. The objective is to provide a framework that imposes minimal constraints on how reconfiguration is actually performed, while providing systematic support for managing the reconfiguration process in terms of configuration stores, performance constraints, and optimization objectives. CMF is a meta-algorithm because specific details of the architecture, the application, and the on-line adaptation algorithms are left unspecified, and can be customized based on the relevant classes of applications and architectures. This meta-algorithm maintains a *current objective* at all times, where the goal is always to improve the current objective without violating any of the previously satisfied constraints. The function *onLineAdaptation* takes an objective metric, a constraint value, and a configuration as inputs, and keeps refining the configuration in an effort to continually improve its quality (as defined in Section 3.1). This function would typically be called within an enclosing loop that performs any system-dependent re-initialization and re-invokes the function immediately after the previous invocation of the function terminates (observe that the function terminates when the current goal is changed).

Pseudocode for the related functions is given in Figure 3. We have implemented CMF, and simulation results pertaining to it are discussed in Section 5.

4.1 Issues related to configuration management

Before proceeding with discussion of our experiments with CMF, we first study some fundamental versions of the problems related to configuration management, discuss their complexity, and relate aspects of them to well-studied problems. Two related problems regarding the size of the configuration store are as follows.

P1. Find the minimum size configuration store and the goals that should be stored in it such that all the relevant goals are covered.

P2. If one has a well-defined measure of "distance" between goals and the goal-pace is a metric space [4], then for a given fixed size configuration store, find the goals whose configurations should be stored such that the sum of the distances of those goals that are not present in the configuration store, from the distance-wise nearest goal present in the configuration store, is minimum.

```
function CMF

/* Global variables accessed: the current goal and the set
of pre-computed configurations, respectively.
*/

global goal $g_c = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K), m_R]$
global configurationStore C

stack $S$ = emptyStack ;
goal $g$, $g_o$;
$g = g_o = g_c$

/* The current optimization metric and the current
constraint to satisfy
*/
objective $objective_c = m_R$
constraint $constraint_c$ = null

$\sigma = \{c \in C | c \text{ satisfies } g\}$
while ($\sigma = \emptyset$) {
      ($g$, $constraint_c$, $objective_c$) = demoteConstraint($g$, $S$)
      $\sigma = \{c \in C | c \text{ satisfies } g\}$
}

/* Select an admissible configuration from $\sigma$ using some
heuristic or specialized, optimal algorithm.
*/
$configuration_c$ = select($\sigma$)

/* Keep trying to refine the current configuration accord-
ing to the current goal until the goal changes ($g_c$ may
change at any time under external control).
*/
while ($g_c = g_o$) {
      while ($constraint_c$ is not satisfied by $configuration_c$) {
            onLineAdaptation($objective_c$, $constraint_c$,
                        $configuration_c$)
      }
      /* Move to the next unsatisfied constraint or to
      the residual objective */
      ($g$, $constraint_c$, $objective_c$) = promoteConstraint($g$, $S$)
}
end function
```

Fig. 2. The CMF framework for goal-driven reconfiguration.

P1 and P2 can be viewed, respectively, in terms of the well-known problems of minimum dominating sets and k-medians. To reduce P1 from the minimum dominating set problem [5], for every vertex in the dominating set problem, instantiate a goal, and for every edge, instantiate a condition that the goal corresponding to the source vertex is acceptable to the goal corresponding to the sink vertex. The problem P1 related to this set of goals and the acceptability relation among goals is equivalent to the given minimum dominating set problem instance. The vertices in the given minimum dominating set problem instance, corresponding to the goals that should be stored in the configuration store (found by solving P1) constitute a minimum dominating set for the given minimum dominating set problem instance. This can be used to show that the problem P1 is NP-hard (see [9] for more details). However, if the acceptability relation is a partial order, then the minimum dominating set can be found in polynomial-time by picking up all the vertices with no incoming edges in the graph of the minimum dominating set problem. This is in accordance with Theorem 1, and further underscores the advantage of using acceptability relations that are partial orders.

If the associated distance function is defined between any two goals and the goal-space is a metric space, then problem P2 can be modeled in terms of the *k-median*

```
function promoteConstraint
input goal $g = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), m_K]$, stack $S$
output goal, constraint, objective
goal $g'$
constraint $v = S$.pop()
objective $m = S$.pop()
constraint $x = S$.pop()
$S$.push($x$)
$g' = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), (m_K, v), m]$
return {$g'$, $x$, $m$)
end function
```

```
function demoteConstraint
input goal $g = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K), m_R]$, stack $S$
output goal, constraint, objective
goal $g'$
$S$.push($m_R$)
$S$.push($c_K$)
$g' = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), m_K]$ .
return ($g'$, $c_K$, $m_K$)
end function
```

Fig. 3. Definition of functions *promoteConstraint* and *demoteConstraint* from Figure 2.

problem [2, 8], as shown in [9]. For the simple case of a two-dimensional goal space, a polynomial-time approximation algorithm with a 3-approximation factor exists for the k-median problem [2].

Configuration management problems P1 and P2 can be viewed as extreme cases in the sense that in one of them we want to cover all feasible goals without considering how large the minimum size configuration store would be (P1), and in the other case, we have a fixed size configuration store and we are trying to find out the maximum number of goals that can be covered using that configuration store even though that number could be much less than the total number of relevant goals (P2). A more elaborate formulation would be one in which we have to pay extra cost for increasing the size of the configuration store, but we would be gaining some additional service by that by being able to store more goals in the configuration store. This way we can explore various trade-offs between the size of the configuration store vs. the number of goals stored in a well-defined way. For the specific case when a distance function is defined between any two goals and the goal-space is a metric space, these trade-offs can be explored by modeling this problem as a facility-location problem [4, 8, 10], as explained in [9]. A polynomial-time algorithm with an approximation guarantee of 1.74 exists for the facility location problem [4].

5. On-line adaptation

In this section, we focus on the metrics of throughput and power consumption, and develop low-complexity, on-line strategies based on heuristics for throughput optimization and power optimization as implementations of the function *onLineAdaptation* in Figure 2. The objective is to demonstrate the efficacy of the CMF model, and show that it can produce efficient tracking of time-varying application requirements.

The approach of taking feedback from the execution of the application makes these on-line methods able to handle even applications with stochastic execution times that have time-varying distributions, in addition to applications with fixed execution times, and applications with stochastic attributes that have stationary distributions. In general, this on-line refinement formulation can thus be viewed as an approach to tracking the dynamics of the goal and the characteristics of the application.

To experiment with CMF, we used a simple heuristic based on load balancing [15] to optimize throughput during online adaptation. Pseudocode for this heuristic is represented by function *adaptThroughput* in Figure 4. In the pseudocode, $moveTask(c, n)$ is a function that chooses n tasks from a maximally loaded processor in a configuration c, and randomly, moves them to appropriate locations on a minimally loaded processor, and returns the modified configuration. Randomization in choosing tasks from the maximally loaded processor provides a low-complexity approach to increase the explored region of the design space and to calibrate the configuration to dynamic application characteristics. The function $executeTr(c, l)$ is a function that executes the application according to configuration c for a time interval of length l, and returns the throughput of the application during that interval. The value of l to use depends on the non-determinacy of the application. We define the non-determinacy of an application in the following way.

Let the number of possible execution times taken by an actor i be denoted by n_i.

We denote the set of n_i possible execution times taken by an actor i as $\{t_{i1}, t_{i2}, \ldots, t_{in_i}\}$. The probability of occurrence of a possible execution time t_{ik} for actor i, is denoted by p_{ik}, for all $k = 1, \ldots, n_i$. The *degree of non-determinacy* λ is a measure of the overall amount of non-determinacy in the application, specifically, in the actor execution times, and is defined as

$$\lambda = \frac{\sum_i \left\{ \sum_{k=1}^{n_i} \left\{ p_{ik}(t_{ik} - t_{i,mean})^2 \right\} \right\}}{\sum_i \{t_{i,mean}\}^2}, \tag{6}$$

where $t_{i,mean}$ denotes the mean execution time of actor i, and is defined as

```
/* This function adapts the given input configuration
      while executing the application.
*/
function adaptThroughput
input configuration $c$
global constant time $timelimit$, time $l$

time $t_{old} = \text{executeTr}(c, l)$
time $t$
configuration $c_{old} = c$
$n = 1$
while ($\text{clock} < timelimit$) {
        $c = \text{moveTask}(c_{old}, n)$
        if (exhausted all $n$-task movements
                without improvement){
                $n = n + 1$
                $c = \text{moveTaskTr}(c_{old}, n)$
        }
        $t = \text{executeTr}(c, l)$
        if($t \geq t_{old}$) {
                $c_{old} = c$
                $t_{old} = t$
                $n = 1$
        }
        $\text{clock} = \text{clock} + l$
}
end function
```

Fig. 4. An online adaptation approach for throughput optimization.

$$t_{i,mean} = \left(\sum_{k=1}^{n_i} t_{ik}\right) / n_i . \quad (7)$$

Generally, the more non-deterministic the application is, the longer it needs to be executed to determine an accurate value of average throughput.

The function *adaptThroughput* returns a configuration that it deems most appropriate for throughput maximization. Note that if moving any single task from the maximally loaded processor to the minimally loaded processor does not improve performance then the heuristic chooses a *pair* of tasks to be moved to another processor. This approach of progressively increasing the number of tasks to be moved continues whenever all combinations for a particular number of tasks have been exhausted. This approach thus attempts to make small low-complexity changes first and if that does not improve performance, the approach gradually reaches towards higher-complexity changes. The higher complexity changes are larger in number than small, low-complexity changes, and help the system in escaping from local minima.

In our experiments, inter-processor communication (IPC) per time unit during the execution is taken as an estimate for relative power consumption. Since IPC consumes relatively large amounts of power, it is a reasonable approximation for comparing the power consumption levels of alternative configurations on a homogeneous multiprocessor. To find a configuration that reduces the power consumption, we use an approach (called *adaptPower*) similar to the *adaptThroughput* approach used for throughput optimization, except that the probability of a task on a maximally loaded processor being transferred to a minimally loaded processor depends upon the IPC associated with that task. The higher the IPC associated with a task, the higher its chances are of being transferred to another processor.

6. Experimental results

An on-line adaptation scheme for refining a given goal is specified in Figure 5, and it is represented as function *onlineAdaptation* in the CMF pseudocode of Figure 2. In Figure 5, the appropriate online optimization strategy, such as the *adaptThroughput* or *adaptPower* approaches discussed above, is selected depending on the current optimization objective and system state. Typically, this strategy will be drawn dynamically from a library of simple, low-complexity techniques.

Table 1 shows the performance of our implementation of CMF using the heuristics developed in Section 5 for throughput optimization and power optimization based on various goals applied to several DSP benchmarks, including fast Fourier transform, filter bank, music synthesis, and measurement applications. The starting configuration that is refined is found by using standard critical path scheduling. The critical path length is computed in terms of average execution times of actors. The set of relevant metrics M for our experiments is $M = \{T, P\}$, where T denotes the average iteration period of the execution and P denotes the average power consumption. Experiments are reported for the following eight goals.

g_1 = {(P, 0.270), (T, 265), (P, 0.250), (T, 0.255), P}

g_2 = {(T, 260), (P, 0.240), T}
g_3 = {P, 0.125), (T, 180), P}
g_4 = {(T, 165), (P, 0.110), (T, 160), P}
g_5 = {(T, 360), (P, 0.160), (T, 355), (P, 0.155), (T, 350), P}
g_6 = {(T, 345), P}
g_7 = {(T, 215), (P, 0.040),T}
g_8 = {(P, 0.053), (T, 215), (P, 0.050), (T, 210), P}

In Table 1, the column titled "Goal" represents the goal that is applied to the application. Also, for a non-negative integer k, column v_k denotes the value of a metric of the best configuration found by the on-line adaptation scheme, after k configurations have been assessed by executing them for some time. For the same experiments that are reported in Table 1, Table 2 shows the times at which different constraints associated with the applied goals are satisfied. For a given goal that is applied to an application, n_i denotes the number of configurations that have been executed in order to assess them before the ith constraint in the applied goal is satisfied. One can see

Applicati on	λ	Goal	Metric	v_0	v_{10}	v_{20}	v_{30}	v_{40}	v_{50}	v_{60}
fft1	0	g_1	T	278	278	278	278	256	254	254
			P	.273	.269	.269	.269	.204	.226	.226
fft1	.359	g_2	T	309	256	251	251	251	252	259
			P	.242	.282	.278	.278	.278	.257	.221
qmf	0	g_3	T	145	242	198	198	186	170	170
			P	.133	.117	.098	.098	.088	.096	.096
qmf	.256	g_4	T	142	164	162	162	153	153	153
			P	.136	.127	.110	.110	.110	.110	.110
karp	0	g_5	T	395	353	346	342	342	342	342
			P	.131	.158	.156	148	.148	.148	.148
karp	.309	g_6	T	450	352	300	342	342	346	346
			P	.115	.155	.159	.151	.151	.148	.148
meas	0	g_7	T	220	212	201	184	184	184	184
			P	.054	.075	.059	.021	.021	.021	.021
meas	.405	g_8	T	185	218	212	212	212	210	196
			P	.064	.018	.037	.037	.037	.019	.040

Table 1. Experimental results for CMF.

```
function onLineAdaptation
input $objective_c$ , $constraint_c$ , $configuration_c$
global goal $g$ , $g_c$ , $g_o$
global constant time $timelimit$
global stack $S$ /* constraint stack */

time $t = 0$
while ($t < timelimit$ and $g_c = g_o$){
      while $constraint_c$ is not satisfied {
            if ($objective_c = throughput$) {
                  adaptThroughput($configuration_c$)
            } else if ( $objective_c = power$ ) {
                  adaptPower($configuration_c$)
            } else if …
                  … /* adapt for other objectives */ …
            }
      }
      ($g$, $constraint_c$, $objective_c$) = promoteConstraint($g$, $S$)
}
end function
```

Fig. 5. On-line adaptation scheme. This is an elaboration of function *onLineAdaptation*, which is called in Figure 2. It is effectively a wrapper for specialized reconfiguration optimizations.

App.	λ	Goal	n_1	n_2	n_3	n_4	n_5
fft1	0	g_1	1	37	39	42	-
fft1	.359	g_2	7	56	-	-	-
qmf	0	g_3	8	48	-	-	-
qmf	.256	g_4	0	13	36	-	-
karp	0	g_5	4	7	9	28	28
karp	.309	g_6	16	-	-	-	-
meas	0	g_7	8	28	-	-	-
meas	.405	g_8	3	17	17	48	-

Table 2. Results for CMF tracking an applied goal.

that in these experiments, CMF is able to meet the constraints specified in the given goals within a reasonable number of configurations.

7. Conclusion

In this paper, we have developed a framework called CMF for on-line adaptation of system-wide configurations of embedded multiprocessors. The objective is to provide a framework that imposes minimal constraints on how reconfiguration is actually performed (i.e., the specific optimization algorithms that are used during off-line and online configuration synthesis), while providing systematic support for managing the reconfiguration process in terms of configuration stores, performance constraints, and optimization objectives. The CMF approach is shown to be effective through analysis and experimental results on several DSP benchmarks, which demonstrate the ability of CMF to systematically adapt system configurations towards progressively better solutions for a variety of goals, even in the presence of significant uncertainties in task execution times.

8. References

1. S. S. Bhattacharyya. *Hardware/software co-synthesis of DSP systems.* In Y. H. Hu, editor, *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, pages 333-378. Marcel Dekker, Inc., 2002
2. T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. *Journal of Design Automation for Embedded Systems*, 3(1):23-58, 1998.
3. M. Charikar and S. Guha. improved combinatorial algorithms for facility location and k-median problems. *Proc. 40th Annual Symposium on Foundations of Computer Science*, 378-388, 1999.
4. F. Chudak, "Improved approximation algorithms for uncapaciateted facility location", In R. E. Bixby, E. A. Boyd and R. Z. Rios-Mercado, eds., *Integer Programming and Combinatorial Optimization*, Springer LNCS Vol. 1412, 180-194, 1998.
5. T. Cormen et al. *Introduction to Algorithms*, McGraw Hill, 2000.
6. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman and company, 1999.
7. J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 24-33, April 1997.
8. K. Jain and V. V. Vazirani, "Approximation algorithms for metric Facility location and k-median problems using the primal-dual scheme and Lagrangian relaxation", *Proc. Foundations of Computer Science, 1999.*
9. S. Lohani and S. S. Bhattacharyya. System synthesis for polymorphous computing architectures. Technical Report UMIACS-TR-2002-12, Institute for Advanced Computer Studies, University of Maryland at College Park, February 2002. Also Computer Science Technical Report CS-TR-4330.
10. D. B. Shmoys, E. Tardos, and K. I. Aardal. Approximation algorithms for facility location problems. *Proc. 29th ACM Symp. on Theory of Computing,* 265-274, 1997.
11. S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors:Scheduling and*

Synchronization, Marcel Dekker, 2000.
12. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, “Baring it all to Software: Raw Machines”, *IEEE Computer,* September 1997, pp. 86-93.
13. S. Wong, S. Vassiliadis, and S. Cotofana. Microcoded reconfigurable embedded processors: Current developments. In *Proceedings of the International Workshop on System Architecture Modeling and Simulation*, pages 207-223, July 2001.
14. E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257-271, November 1999.
15. A. Y. Zomaya, “Parallel and Distributed Computing: The Scene, the Props, the Players,” *Parallel and Distributed Computing Handbook,* A.Y. Zomaya, ed., pp. 5-23, New York: McGraw-Hill, 1996.

Customising Flexible Instruction Processors: A Tutorial Introduction

Shay Ping Seng[1], Wayne Luk[1] and Peter Y.K. Cheung[2]

[1] Department of Computing, Imperial College, London, UK.
[2] Department of EEE, Imperial College, London, UK.

Abstract. This paper presents a tutorial about the Flexible Instruction Processor (FIP) methodology which facilitates trade-offs between area, performance and functionality of instruction processor designs, both at compile time and at run time. We explore the customisation of FIPs and discuss the use of FIPs to target the design space between off-the-shelf instruction processor designs that provide flexible computation, and custom hardware designs that provide high performance. We demonstrate how customisation enables FIPs to perform competitively with current main-stream processors and also with dedicated hardware, in certain applications such as CaffineMark Benchmarks and AES encryption.

1 Introduction

Software implementations on standard commercial processors often provide good performance, a wide range of functionality and an effective means of targeting evolving standards. However, they tend to lose efficiency when dealing with non-standard operations and non-standard data that are not supported by their instruction set [21]. Direct hardware implementations tend to provide the best performance for a given application, but lack the flexibility of an instruction processor. When designs are implemented on FPGAs, reconfiguration can be used to gain functionality, thereby providing flexibility to direct hardware designs. This is at the expense of performance, since the time required for reconfiguration can be long and will become longer due to the increasing density of FPGAs.

The resources afforded by large programmable devices makes implementing instruction processors on FPGAs increasingly attractive [6]. Customisation of processors implemented on configurable logic provides a way to balance the trade-offs between direct hardware and software implementations. One way of supporting customisation is to augment an instruction processor with programmable logic for implementing custom instructions.

This paper presents a tutorial introduction about the Flexible Instruction Processor (FIP) methodology which facilitates trade-offs between area, performance and functionality, both at compile time [19] and at run time [20]. In particular, we explore the customisation of FIPs and discuss the use of FIPs to target the design space between off-the-shelf instruction processor designs that provide flexible computation, and custom hardware designs that provide high performance. We demonstrate how customisation enables FIPs to perform competitively with current main-stream processors and also with dedicated hardware, for applications such as CaffineMark Benchmarks and AES encryption.

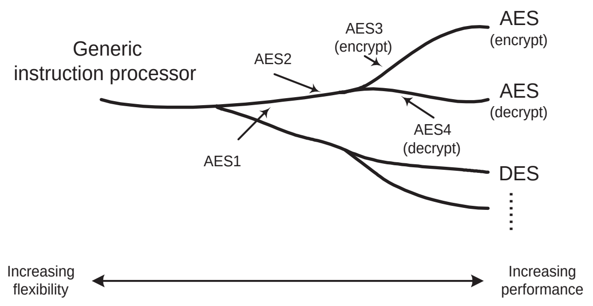


Fig. 1. FIP design spectrum. On the left of the spectrum lies the generic instruction processor, such as one that executes Java bytecodes. AES1 and AES2 are FIPs optimised for AES operation. AES3 and AES4 are further optimised respectively for AES encryption and decryption. FIPs on this spectrum have similar area constraints.

2 FIP Design Spectrum

Standard general-purpose instruction processors are highly optimised and are implemented in custom VLSI technology. They are fixed in architecture, and each represents a point in a spectrum of possible implementations. Our FIP methodology provides a way of traversing the design spectrum to create customised processors that are tuned for specific applications, at compile time and at run time.

FIPs provide a well-defined control structure that facilitates varying the degree of sharing for system resources. This allows critical resources to be increased as demanded by the application domain, or eliminated if not used. FIPs also provide a systematic method for supporting customisation by allowing user-designed hardware to be accommodated as new instructions. These design-time optimisations provide a means of tailoring an instruction processor to a particular application or to applications for a specific domain, such as data encryption.

Direct hardware implementations provide fast performance but once they have been manufactured and deployed, there is little scope for improvement. Instruction processors, on the other hand, provide a solution that is easily upgradable and flexible. However this flexibility is often provided at the expense of performance. FIPs provide a way to explore the design space between these two extremes. For instance, custom instructions can be included into a design to speed up their operation, at the expense of increasing area and power consumption.

To illustrate our approach, consider designs that we have developed to support AES (Advanced encryption standard) [13] for data encryption and decryption. These designs have similar area constrains. AES1 in Figure 1 is a FIP implementation of the AES algorithm. It has been customised by removing hardware associated with unused opcodes in the generic instruction processor. However AES1 does not contain any custom instructions. AES2 has three custom instructions that will speedup both encryption and decryption. AES2 shows better performance for both encryption and decryption when compared to AES1. The new custom instructions replace the functionality of some opcodes. The opcodes no longer in use are removed to provide area for the custom instructions. Hence AES2 is less flexible than AES1, in that some programs executable on AES1 may no longer be executable on AES2. AES3 is a further specialisation that improves the performance of AES encryption: a new custom instruction that replaces the inner loop for encryption has been introduced. This provides a four-fold speedup over AES2; however, the three custom instructions introduced in AES2 have to be removed to make space for this new instruction. So while encryption speed is improved, decryption speed suffers. AES4 specialises the decryption routine to give a five-fold improvement over AES2 but with the same kind of trade-offs as AES3.

In contrast to FIPs, each conventional instruction processor occupies one point in the design spectrum shown in Figure 1. Once deployed, these processors cannot adapt itself to suit run-time conditions, unlike FIPs.

3 Related Work

The viability of implementing instruction processors on FPGAs has been demonstrated by Altera [2] and Xilinx [28]. Many techniques and tools for customising instruction processors have been reported. This section outlines a small subset and organises them into three categories: fixed processors coupled with configurable logic, partially configurable processors, and fully configurable processors.

The PRISC [18], Chimaera [7], ConCISe [9] and DISC [26] architectures are examples of systems that couple a fixed processor core with field programmable hardware. PRISC provides customisation in the form of programmable functional units. The goal of PRISC is to augment the performance of the RISC microprocessor by allowing programmable functional units to be pipelined at a granularity that is smaller than the existing cycle time. The Chimaera system is similar to PRISC. The Chimaera reconfigurable functional unit can be configured to implement a 4-LUT, two 3-LUTs or a 3-LUT or a carry chain computation. However, Chimaera logic cells do not contain latches or flip-flops and require results to be stored back to the register file.

The ConCISe system features a CPLD-based reconfigurable functional unit and a system to encode multiple custom instructions in a single reconfigurable unit. The objective of this technique is to reduce the time for reconfiguration. Custom instructions implementable in ConCISe are limited to combinational logic. These three systems, PRISC, Chimaera and ConCISe, provide compilation tools that attempt to automatically generate mappings for the reconfigurable logic. Custom instructions tend to be fine-grain and relatively small, due in part to the difficulty of the matching problem and the size of the programmable fabric available. Like the systems mentioned above, DISC consists of a main processor coupled with reconfigurable functional units. The DISC system requires custom instructions to be identified and programmed manually. The main focus of the DISC system is in the handling of the loading of custom instructions. DISC treats the reconfigurable logic as a cache, with a miss in the cache resulting in an automatic stall and the loading of the required custom instruction. Several vendors [1, 24, 27] are also offering a route to implementations featuring fixed processor cores interfaced to programmable logic.

NIOS [2], MicroBlaze [28] and Xtensa [23] are configurable processors that implement a fixed instruction set, but allows the implementation of the processor to be customised to a limited degree. NIOS and MicroBlaze are designed to be configured and run on an FPGA, while Xtensa is designed to be implemented on ASICs. Design tools for these systems help the processor designer to customise a processor at design time. The NIOS and Xtensa system support custom instructions but they require the custom instructions to be hand-crafted.

CRISP [3] and BUILDABONG [16, 22] are projects that explore methods for prototyping instruction sets and application specific processor designs. CRISP provides a template for reconfigurable instruction set processors to be described. The BUILDABONG project divides its goals into four phases: architecture description and composition, simulation, compiler generation and optimal architecture and compiler-codesign.

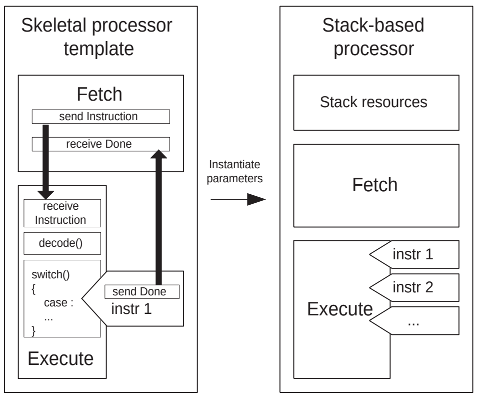
The FIP approach is based on a technique advocated by Page [15] where instruction processor designs are captured as parallel computer programs. Our work includes three main themes: (a) techniques for customising the design of FIP architectures, (b) a tool framework for generating and optimising executable code for FIPs, and (c) extensions of the above to cover run-time reconfigurable designs. Although FIPs can target ASICs, our focus is on targeting FPGAs. Implementing FIPs on FPGAs allow us to explore the possibility of adapting FIPs to the run-time characteristics of a system over a period of time.

4 Flexible Instruction Processors

FIPs consist of a processor template and a set of parameters [19]. Different processor implementations can be produced by varying the template parameters. FIP templates provide a general structure for creating processors of different styles: for instance stack-based or register-based processors. The processor templates can be further enhanced with features found in modern high-performance processors, such as superscalar and pipelined architectures. Various Java Virtual Machines [10] and MIPS style processors [12] have been implemented. Our work is intended to provide a general method for creating processors with different styles.

When compared with a direct hardware implementation, instruction processors have the additional overheads of instruction fetch and decode. However, there are also many advantages.

- FIPs allow customised hardware to be accommodated as new instructions. This combines the efficient and structured control path associated with an instruction processor with the benefits of hand-crafted hardware. The processor and its associated opcodes provide a means to optimise control paths



```
#include stack
#include top_of_stack_reg
#include program_counter
chan fetchchan;

par{
{// -- Fetch module --
Fetch_from_memory(instr);
//send instr to Execute
fetchchan ! instr;
pc++;
...}

{// -- Execute module --
//recieve instr from Fetch
fetchchan ? instr;
switch (decode(instr))
  {
  #include stack_instr1
  #include stack_instr2
  ...}
}
}
```

Fig. 2. A skeletal processor template. The Fetch module fetches an instruction from external memory and sends it to the Execute module, which waits until the Execute module signals that it has completed updating shared resources, such as the program counter. This diagram also shows the instantiation of a skeletal processor into a stack processor, and the Handel-C description of the stack processor. The `par` construct allows statements to be executed in parallel. The statement `c?x` assigns the value read from the channel `c` to the variable `x`, while the statement `c!e` writes the expression `e` to the channel `c`.

through optimising compilers. Non-standard datapath sizes can also be supported.

- Critical resources can be increased as demanded by the application domain, and eliminated if not used. Instruction processors provide a structure for these resources to be shared efficiently, and the degree of sharing can be determined at run time.
- Our FIP approach enables different implementations of a given instruction set with different design trade-offs. It is also possible to relate these implementations by transformation techniques [15], which provide a means of verifying non-obvious but efficient implementations.
- FIPs enable high-level data structures to be easily supported in hardware. Furthermore, they help preserve current software investments and facilitate the prototyping of novel architectures, such as abstract machines for exact real arithmetic and declarative programming [14].

FIPs are assembled from a processor template with modules connected together by communicating channels. The template can be used to produce different styles of processors, such as stack-based or register-based styles. The parameters for a template are selected to transform a skeletal processor into a processor suited for its task (Figure 2). Possible parametrisations include addition of custom instructions, removal of unnecessary resources, customisation of data and instruction widths, optimisation of opcode assignments, and varying the degree of pipelining.

When a FIP is assembled, required instructions are included from a library that contains implementations of these instructions in various styles. Depending on which instructions are included, resources such as stacks and different decode units are instantiated. Channels provide a mechanism for dependencies between instructions and resources to be mitigated.

The FIPs in our framework are currently implemented in Handel-C, a C like language for hardware compilation supported by the DK 1 design suite [4]. Handel-C has been chosen because it keeps the entire design process at a high level of abstraction, which benefits both the design of the processor and the inclusion of custom instructions. Handel-C also provides a quick way to prototype designs. Our focus is to provide FIPs that are customised for specific applications, particularly light-weight implementations for embedded systems. Using a high-level language like Handel-C simplifies the design process by having a single abstract description. A high-level language can also provide a mechanism for demonstrating the correctness of the FIP [8, 15].

5 Customising FIPs

Modern instruction processors contain many features to enhance execution efficiency; examples include superscalar, VLIW and EPIC architectures. The techniques used by VLIW and EPIC architectures to reduce the ratio of instruction fetches to executions can be incorporated into FIPs. Additionally FIPs provide a mechanism for more complex performance trade-offs to be made. Execution efficiency can be traded off with area, power consumption and functionality.

Custom instructions reduce the ratio of the time for instruction fetch and the time for instruction execution, increasing the performance of the instruction processor. Additional resource is introduced so area is increased. However due to the increase in execution efficiency, it may be possible to lower the clock speed of the processor while maintaining an acceptable level of performance. Examples of such trade-offs will be given in Section 7.

FIP implementation	Area (gates and latches)	Cycles
(a) Sequential no multiplier	500	81
(b) Sequential with multiplier	606	39
(c) Sequential with custom instruction	687	20
(d) Pipelined with custom instruction	938	14

Table 1. Various FIPs with different area-efficiency trade-offs for the sum of squares computation. Area results are taken from technology independent estimates provided by the DK 1 design suite. FIP (a) is an accumulator style processor that implements multiplication with repeated additions. FIP (b) is the same as (a) except that it has a multiplier unit. FIP (c) has dedicated resources to calculate the sum of two squares. FIP (d) is a pipelined version of (c).

Let us consider a simple example. Table 1 shows FIPs with different trade-offs for calculating the sum of the square of two numbers. FIP (a) is small but relatively inefficient. FIP (b) is larger since it contains a multiplier unit. Including custom instructions and pipelining also greatly improves the performance, but also increase the area. Custom instructions can be hand-crafted or generated automatically, by directly connecting up the data path of the sequence of opcodes that make up the sum of squares function. As Table 1 shows, higher performance can be achieved at the expense of area. If however area is constrained, FIP functionality will have to be reduced, making the FIP less flexible. This would thus correspond to a right movement in the FIP design spectrum in Figure 1. Section 7 contains more such examples.

Different styles of custom instructions can be incorporated into FIPs. Examples of such custom instructions include look-up table based instructions and streaming style instructions [20].

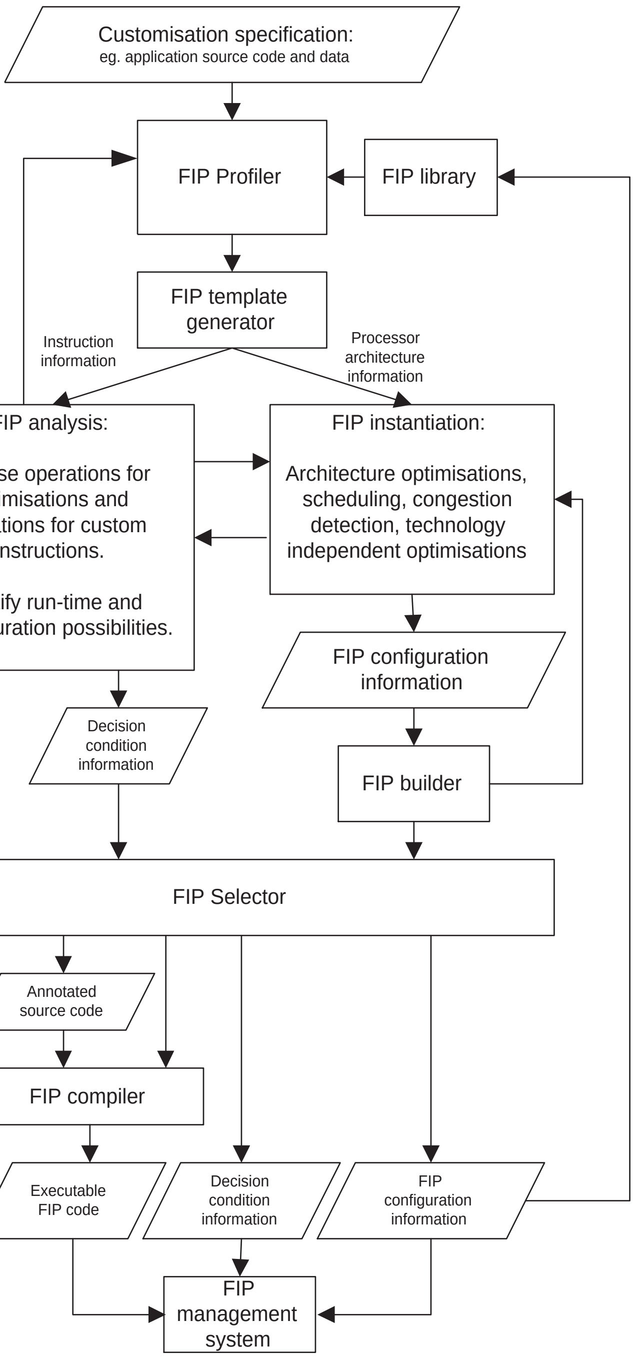
6 Compilation Strategy

Our design flow is described in Figure 3. The input to the design environment is a specification of the application. The application specification can take several forms; C or Java for instance. At this stage, user design options can be included to constrain speed, area or resources used.

The FIP library contains templates of different processors, such as JVM or MIPS for instance. The profiling step collects data such as frequency of certain combination of opcodes and resources required. The initial set of parameters derived from the user's specification is augmented by information gathered by the FIP profiler. The FIP template generator creates an initial FIP.

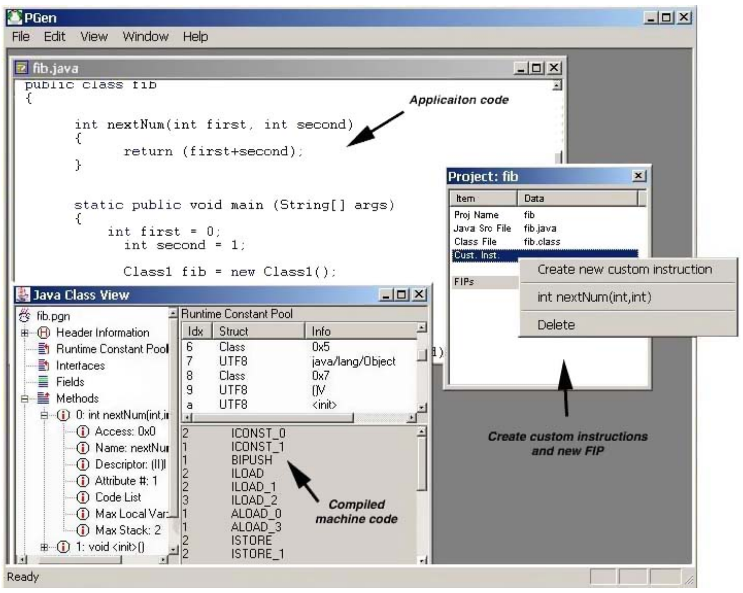
At this point, the design flow is split into an analysis step and FIP instantiation step. The analysis step involves analysing sharing possibilities and introducing custom instructions. The custom specification is profiled and code candidates for implementation into custom instructions are analysed. Run-time reconfigurable possibilities are also explored. The FIP instantiation stage involves architecture optimisations on congestion, scheduling, speed, area and latency. Custom instructions selected by the analysis step is also instantiated. Technology-specific optimisations, such as using vendor-provided macros or technology-specific features like block RAMS, can also be applied at this level. The completion of these two steps results in source code for the application, decision condition information (to detail when to configure to this FIP) and the FIP configuration information. FIP instantiation and analysis can be iteratively employed to produce different variations of FIPs with different characteristics to achieve good speed-area trade-offs. The decision condition information is used by the run-time environment to decide if this FIP is required for execution.

The FIP selector stage selects one or more FIP implementation and compiles the source code into code executable by that FIP. The executable code, decision condition information and FIP configuration information is then provided to the FIP management system for execution.



Customisation specification:
eg. application source code and data
FIP Profiler
FIP library
FIP template generator
Instruction information
Processor architecture information
FIP analysis:
Analyse operations for optimisations and operations for custom instructions.
Identify run-time and reconfiguration possibilities.
FIP instantiation:
Architecture optimisations, scheduling, congestion detection, technology independent optimisations
FIP configuration information
Decision condition information
FIP builder
FIP Selector
Annotated source code
FIP compiler
Executable FIP code
Decision condition information
FIP configuration information
FIP management system

Fig. 3. FIP design flow. This diagram shows the steps involved in producing a FIP tuned to a customisation specification.



```
public class fib
{

        int nextNum(int first, int second)
        {
                return (first+second);
        }


        static public void main (String[] args)
        {
            int first = 0;
              int second = 1;

              Class1 fib = new Class1();
```

Item	Data
Proj Name	fib
Java Src File	fib.java
Class File	fib.class
Cust. Inst.	
FIPs	

Idx	Struct	Info
6	Class	0x5
7	UTF8	java/lang/Object
8	Class	0x7
9	UTF8	()V
a	UTF8	<init>

```
2    ICONST_0
1    ICONST_1
1    BIPUSH
2    ILOAD
2    ILOAD_1
3    ILOAD_2
1    ALOAD_0
1    ALOAD_3
2    ISTORE
2    ISTORE_1
```

Fig. 4. The PGen design tool allows a Java program to be compiled and analysed. It also allows custom instructions to be created. FIPs and their associated executable code can also be generated.

Figure 4 shows PGen, an implementation of our design flow. The window labelled "Application code" shows a Java program as the custom specification. The Java code can be compiled and inspected with the Java Class View window. The Java Class View tool analyses the compiled code and provides information such as the number of JVM opcodes used, the frequency of their use, the opcodes used by various functions in the class, and the data stored in the JVM's constant pool. This step corresponds to the analysis block in Figure 3. This information is used to instantiate the FIP. The right pop-up box shown next to the project window, shows PGen suggesting the function nextNum as a candidate for implementation as a custom instruction. Once the user is satisfied with the design, a FIP and its associated executable code is produced.

The run-time environment is responsible for the execution and management of the system. It maintains a database of available FIPs, their associated executable code and a decision condition library. If configuration occurs too frequently, the overall performance of the system can suffer. The run-time environment provides a way to fine tune the frequency of reconfiguration. For instance, it can decide that a more efficient FIP cannot be introduced if the reconfiguration time is unacceptable. During execution, a FIP can keep track of run-time statistics, such as the number of times that user-specified functions are called, or the most frequently used opcodes. These run-time statistics can be used to adapt the FIP. Further information about the run-time environment for FIP can be found elsewhere [20].

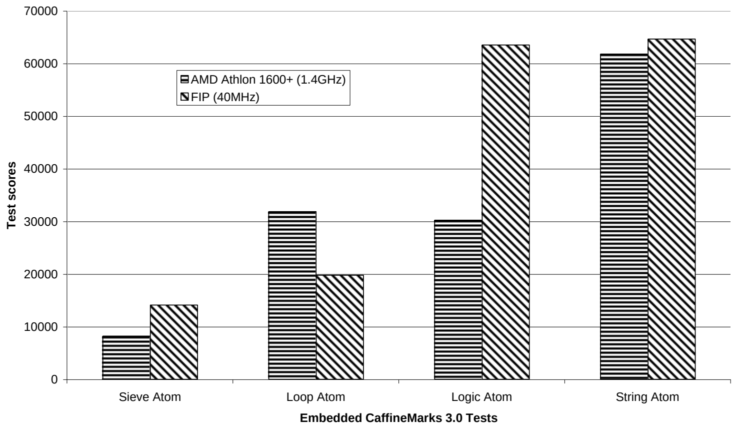


Fig. 5. Caffine Mark 3.0 results. Higher scores are better. On the AMD machine, the benchmark is executed on Sun Microsystem's Hot Spot JVM version 1.3.1_02.

7 Implementation and Evaluation

This section describes various implementations of FIPs for the Java Virtual Machine and for implementing the AES algorithm, and compares their performance against general-purpose processors and custom hardware.

7.1 CaffineMark 3.0 Benchmarks

The embedded CaffineMarks 3.0 [17] Java benchmark is a set of tests used to benchmark performances of JVMs in embedded devices. We have compiled four of the six benchmarks in this set. The Sieve Atom implements a classic sieve of Eratosthenes to find prime numbers. The Loop Atom uses sorting and sequence generation to measure optimisation of loop instructions. The Logic Atom tests the speed of decision-making instructions. The String Atom measures the efficiency of string manipulation and finding substring within strings. The other two tests that have not yet been implemented involve floating-point calculation and method invocation speed testing.

Figure 5 shows the performance of a FIP-JVM (clocked at 40MHz) compared with a JVM running on an AMD Athlon XP 1600+ (1.4GHz) machine. The FIP contains custom instructions that help accelerate the execution of the benchmarks. The FIP implementation, despite running on a slower clock, out performs the conventional processor on all but one of the test. For that test, the Loop Atom, the FIP when clocked at 65MHz can achieve the same performance as the conventional processor.

7.2 AES

We illustrate our approach with another example using the AES (Rijndael) algorithm [13], an iterated block cipher with variable block and key length.

We have two implementations. Our first implementation, FIP (i), is augmented by a single custom instruction that directly connects the data paths for the individual component transformations. This achieves an encryption of 128 bits of data with a 128-bit key in 99 cycles.

The AES specification suggests that the AES can be accelerated by unrolling several of the AES functions into look-up tables. Our second implementation, FIP (ii), utilises this method and achieves an encryption of 128 bits of data with a 128-bit key in 32 cycles.

Implementations	Cycles/Block	Hardware resources	Mbps/MHz	Flexible
Software[5] (C/C++)	340		0.4	Yes
FIP (i)	99	1770 Slices 2 BRAMs	1.3	Yes
FIP (ii)	32	1393 Slices 10 BRAMs	4	Yes
Hardware[25] (Spartan II 100-6)	11	460 Slices 10 BRAMs	11.5	No
Hardware[11] (Virtex-E 812-8)	1	2679 Slices 82 BRAMs	129.6	No

Table 2. Various AES implementations. Blocks are 128 bits with 128 bit keys. The C/C++ implementation is written for the Pentium family of processors. FIP implementations are written in Java and run on a sequential version of the JVM implemented on a Spartan II 300E-6. The hardware implementation on the spartan is latency optimised and performs at 0.52 Gbps (45MHz). The hardware implementation on the Virtex-E runs at a data rate of 7 Gbps (54MHz).

Table 2 compares different implementations of the AES algorithm. The fastest reported C/C++ implementation, by Gladman [5], achieves an encryption speed of about 350Mbps on a 933MHz Pentium 3.

Running at 40MHz, FIP(i) encrypts at 51.7Mbps and FIP(ii) at 160 Mbps. FIP(ii) performs ten times better than software, in a Mbps per MHz comparison. Hand-placed hardware implementations provide good performance, but cannot be used for general computations. This flexibility has been compromised to improve performance. However these hardware implementations can be incorporated into FIPs as custom instructions [20].

8 Concluding Remarks

The FIP approach offers a framework for trading off speed, flexibility and area. It provides the flexibility afforded by instruction processors, and custom instructions can be introduced to improve performance. Our proposed design-time and run-time environments provide a means of customising these processors and compiling code for them. They also provide a mechanism for these processors to adapt to run-time conditions, depending on usage patterns.

Current and future work includes improving the efficiency of our FIP implementations, and refining the tools for automating the support for custom instructions.

Acknowledgements
The support of Celoxica Limited and the UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599) is gratefully acknowledged.

References

1. Altera Corporation. *Excalibur Embedded Processor Solutions.* http://www.altera.com/html/products/excalibur.html.
2. Altera Corporation. *Nios Embedded Processor System Development.* http://www.altera.com/products/devices/nios/nio-index.html.
3. P. Op de Beeck et al. CRISP: A template for reconfigurable instruction set processors. In *Field Programmable Logic and Applications*. Springer, LNCS 2147, 2001.
4. Celoxica. *DK 1 Design Suite.* http://www.celoxica.com.
5. B. Gladman. *Implementations of AES (Rijndael) in C/C++ and Assembler.* http://fp.gladman.plus.com/cryptography_technology/rijndael/.
6. J. Gray. Building a RISC system in an FPGA. In *Circuit Cellar: The magazine for computer applications*, pp. 20–27. March 2000.
7. S. Hauck. The chimaera reconfigurable functional unit. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, pages 87–97, 1997.
8. J. He, G. Brown, W. Luk and J. O'Leary. Deriving two-phase modules for a multi-target hardware compiler. In *Proc. 3rd Workshop on Designing Correct Circuits*. Springer Electronic Workshop in Computing Series, 1996.
9. B. Kastrup, A. Bink and J. Hoogerbrugge. ConCISe: A compiler-drive CPLD-based instruction set accelerator. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*. IEEE Computer Society Press, 1999.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Ed.)*. Addison-Wesley, 1999.
11. M. McLoone and J. McCanny. Single-chip FPGA implementation of the Advanced Encryption Standard algorithm. In *Field Programmable Logic and Applications*, LNCS 2147. Springer, 2001.
12. MIPS Technologies Incorporated. MIPS processors. http://www.mips.com.
13. National Institute of Standards and Technology. *Advanced Encryption Standard.* http://csrc.nist.gov/encryption/aes.
14. C. North. Graph reduction in hardware. Master's thesis, Oxford University, 1992.
15. I. Page. Automatic design and implementation of microprocessors. In *Proc. WoTUG-17*, pages 190–204. IOS Press, 1994.
16. Fachgebiet Datentechnik, Universität Paderborn. *The BUILDABONG project.* http://www-date.upb.de/RESEARCH/BUILDABONG/buildabong.html.
17. Pendragon Software Corporation. *CaffineMark 3.0 Java Benchmark.* http://www.pendragon-software.com/pendragon/cm3/index.html.
18. R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, 1994.
19. S. Seng, W. Luk and P. Cheung. Flexible Instruction Processors. In *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, 2000.
20. S. Seng, W. Luk and P. Cheung. Run-time Adaptive Flexible Instruction Processors. To be published in *Field Programmable Logic and Applications*, 2002.

21. H. Styles and W. Luk. Customising graphics applications: techniques and programming interface. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*. IEEE Computer Society Press, 2000.
22. J. Teich and R. Weper. A joined architecture/compiler design environment for ASIPs. In *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2000.
23. Tensilica Incorporated. *Xtensa Configurable Processor*. http://www.tensilica.com/technology.html.
24. Triscend Corporation. *The Configurable System on a Chip*. http://www.triscend.com/products/Index.html.
25. N. Weaver and J. Wawrzynek. *Very high performance, compact AES implementations in Xilinx FPGAs*. http://www.cs.berkeley.edu/∼nweaver/sfra/rijndael.pdf.
26. M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, pp. 99–107. IEEE Computer Society Press, 1995.
27. Xilinx Incorporated. *PowerPC Embedded Processor Solution*. http://www.xilinx.com.
28. Xilinx Incorporated. *MicroBlaze Soft Processor*. http://www.xilinx.com.

Entropy Decoding on TriMedia/CPU64

Mihai Sima[1,2], Evert-Jan Pol[2], Jos T.J. van Eijndhoven[2],
Sorin Cotofana[1], and Stamatis Vassiliadis[1]

[1] Delft University of Technology, Department of Electrical Engineering,
Mekelweg 4, 2628 CD Delft, The Netherlands,
{M.Sima,S.D.Cotofana,S.Vassiliadis}@et.tudelft.nl
[2] Philips Research Laboratories, Department of Information and Software Technology,
Professor Holstlaan 4, 5656 AA Eindhoven, The Netherlands,
{evert-jan.pol,jos.van.eijndhoven}@philips.com

Abstract. The paper describes a software implementation of an MPEG–compliant Entropy Decoder on a TriMedia/CPU64 processor. We first outline entropy decoding basics and TriMedia/CPU64 architecture. Then, we describe the reference implementation of the entropy decoder, which consists mainly of a software pipelined loop. On each iteration, a set of look-up tables partitioning the Variable-Length Codes (VLC) table defined by the MPEG standard are accessed in order to retrieve the *run-level* pair, or detect an *end-of-block* or *error* condition. An average of 21.0 cycles are needed to decode a DCT coefficient according to this reference implementation. Then, we focus on software techniques to optimize the entropy decoding software pipelined loop. In particular, we propose a new way to partition the VLC table such that by exposing the loop prologue to the compiler, testing each of the *end-of-block* and *error* conditions within the prologue becomes superfluous. This is based on the observation that either an *end-of-block* or *error* condition will never occur within the first table look-up. For the proposed implementation, the simulation results indicate that an average of 16.9 cycles are needed to decode a DCT coefficient. That is, our entropy decoder is more than 20% faster than its reference counterpart.

1 Introduction

The introduction of digital audio and video was the starting point of multimedia because it enabled audio and video, as well as text, figures, and tables, to be used in a digital form in a computer and be held in the same manner. However, digital audio and video require a tremendous amount of information bandwidth unless compression technology is used, which in turn calls for a large amount of processing. For example, National Television Systems Committee (NTSC) resolution MPEG-2 [1] decoding requires more than 400 MOPS, and 30 GOPS are required for encoding.

TriMedia/CPU64 is a VLIW core targeted for real-time processing of multimedia streams [2]. Although its processing power allows significant processing of video data, the VLIW core itself was intended to be integrated on-chip with a set of hardwired co-processors which can perform other tasks with stringent real-time requirements in parallel. An example of such co-processor is the Variable-Length Decoder (VLD) [3].

One of the drawbacks of the hardwired solution is the lack of flexibility, since a different full-custom circuit is needed for each particular task. Software programmability ensures that a single device can be applied in a range of different products and can adapt to quickly evolving standards in the media domain. Therefore, a software solution which can provide the needed performance is always preferred to the hardware solution.

When the application exhibits data and instruction-level parallelisms, TriMedia/CPU64 has proved significant speed-up over previous TriMedia families [4]. However, the speed-up is not so high when parallelism is not available. Entropy decoding [5, 6] consists of Variable-Length Decoding (VLD) followed by a Run-Length Decoding (RLD), both VLD and RLD being sequential tasks. Due to data dependency, entropy decoding is an intricate function on TriMedia, since a VLIW architecture must benefit from instruction level parallelism in order to be efficient.

An entropy decoder implementation on TriMedia/CPU64 which can decode a Discrete Cosine Transform (DCT) coefficient in 21 cycles has been proposed by Pol [7]. The VLD is implemented as a repetitive look-up into the Variable-Length Codes (VLC) table defined by MPEG standard, where each iteration analyzes a fixed-size chunck of bits. When a coefficient is completely decoded, a *run-level* pair is generated, otherwise an offset into the VLC table is generated. By employing software pipeline optimization techniques, run-length decoding for the previous decoded symbol is carried out simultaneously with the variable-length decoding of the current symbol.

In this paper we demonstrate that significant improvement over the reference solution is possible if four optimizations are used:

1. partitioning the VLC table in such a way that by exposing the prologue of the software pipeline loop to the compiler, an *end-of-block* symbol or *error* will never be encountered within the prologue;
2. using an extended *barrel-shift* TriMedia-specific operation;
3. storing the lookup tables in such way that all the fields (*run*, *level*, *table_offset*, etc) are each located within the boundaries of a byte. This way, the extraction of each and every such field can be done in a single cycle by TriMedia–specific operations;
4. using variable chunck size, in order to reduce the total size of the tables.

The testing database for our entropy decoder consists of a number of pre-processed MPEG conformance strings from which all the data not representing DCT coefficients have been removed. Therefore, such strings include only *run-level* and *end-of-block* symbols. The simulations carried out on a TriMedia/CPU64 cycle accurate simulator indicate that 16.9 cycles are needed to decode a DCT coefficient with the proposed implementation. That is, our entropy decoder is 20% faster than its reference counterpart.

As an evaluation of the absolute performance of the entropy decoder we propose, we would like to mention some figures claimed by our competitors: 33 cycles per coefficient which exploits SIMD–type operations of a Pentium processor with MultiMedia eXtension (MMX) are claimed by Ishii *et al.* [8], and 26 cycles per coefficient on an TMS320C80 media video processor are claimed by Bonomini *et al.* [9].

The paper is organized as follows. Section 2 gives some background information concerning MPEG compression standard and TriMedia/CPU64 architecture. Entropy decoder implementation issues are presented in Section 3. The experimental framework and results are presented in Section 4. The final section concludes the paper.

2 Background

The MPEG standard [6, 10] uses a large number of compression techniques to decrease the amount of data. Data compression is the reduction of redundancy in data representation, carried out to decrease data storage requirements and data communication costs.

A typical video codec system is presented in Figure 1 [5, 6]. The lossy source coder performs filtering, transformation (such as Discrete Cosine Transform (DCT), subband decomposition, or differential pulse-code modulation), quantization, etc. The output of the source coder still exhibits various kinds of statistical dependencies. The (loseless) entropy coder exploits the statistical properties of data and removes the remaining redundancy after the lossy coding.

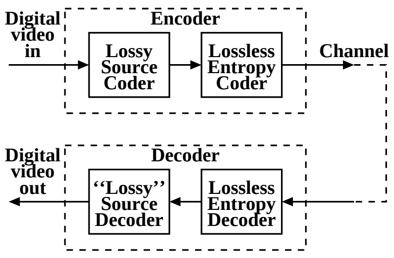


Fig. 1. A generic video codec.

In MPEG, the couple DCT + Quantization is used as a lossy coding technique. The DCT algorithm processes the video data in blocks of 8×8 pixels, decomposing each block into a weighted sum of amplitudes of 64 spatial frequencies. At the output of DCT, the data is also organized as 8×8 blocks of coefficients, each coefficient representing the contribution of a spatial frequency for the video block being analyzed. Since the human eye cannot readily perceive high frequency activity, a quantization step is then carried out. The goal is to force as many DCT coefficients as possible to zero within the boundaries of the prescribed video quality. Then, a zig-zag operation transforms the matrix into a vector of coefficients which contains large series of zeros. This vector is further compressed by an Entropy Coder which consists of a Run-Length Coder (RLC) and a Variable-Length Coder (VLC). The RLC represents consecutive zeros by their run lengths; thus the number of samples is reduced. The RLC output data are composite words, referred to as *symbols*, which describe a *run-level* pair. The *run* value indicates the number of zeros by which a (non-zero) DCT coefficient is preceeded. The *level* value represents the value of the DCT coefficient. When all the remaining coefficients in a vector are zero, they are all coded by the special symbol *end-of-block*. Variable length coding is a mapping process between *run-level*/*end-of-block* symbols and *variable length codewords*, which is carried out according to a set of tables defined by the standard. Not every run-level pair has a variable length codeword to represent it, only the frequent used ones do. For those rare combinations, an *escape* code is given. After an *escape* code, the run- and level-value are coded using fixed length codes.

In order to achieve maximum compression, the coded data does not contain specific guard bits separating consecutive codewords. As a result, the decoding procedure must recognize the *code-length* as well as the symbol itself. Before decoding the next symbol, the input data string has to be shifted by a number of bits equal to the decoded code length. These are recursive operations that generate true-dependencies.

Subsequently, we will focus on the entropy decoding, i.e., on the operation inverse to entropy coding. We will briefly present some theoretical issues connected to variable-length decoding and run-length decoding.

2.1 Entropy Decoder

In MPEG, the entropy decoder consists of a **Variable-Length Decoder** (VLD) followed by a **Run-Length Decoder** (RLD). The input to the VLD is the incoming bit stream, and the output is the decoded symbols. As depicted in Figure 2, a VLD is a system with feedback, whose loop contains a *Look-Up Table* (LUT) on the feed-forward path and a *bit parser* on the feedback path. The LUT receives the variable-length code itself as the address [11] and outputs the decoded symbol (*run-level* pair or *end_of_block*) as well as the codeword length, *code_length*. In order to determine the starting position of the next codeword, the *code_length* is fed back to an accumulator and added to the previous sum of codeword lengths, ***accumulated code_length***. The bit parsing operation is completed by the *barrel-shifter* (or *funnel-shifter*) which shifts out the decoded bits.

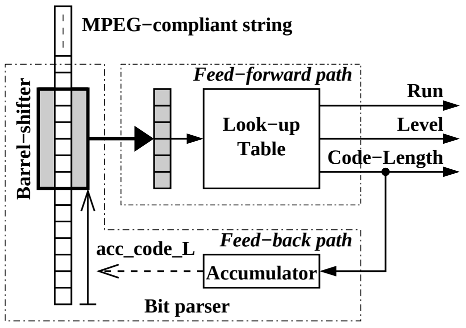
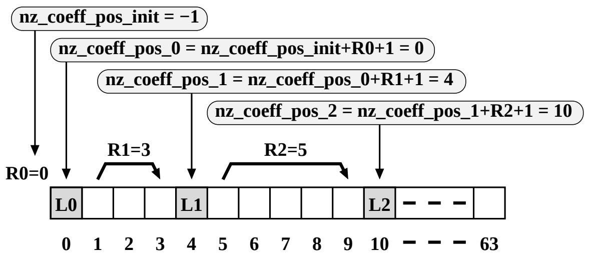


Fig. 2. Variable-length decoding principle.

In connection with the hardware complexity, we would like to note that the longest codeword excluding Escape has 17 bits. Therefore, the LUT size reaches $2^{17} = $ $= 128$ K words for a direct mapping of all possible codewords. Regarding the performance of a variable-length decoder, it is worth mentioning that the throughput of a VLD is bounded by a value inverse to the latency of the loop [12].



nz_coeff_pos_init = −1
nz_coeff_pos_0 = nz_coeff_pos_init+R0+1 = 0
nz_coeff_pos_1 = nz_coeff_pos_0+R1+1 = 4
nz_coeff_pos_2 = nz_coeff_pos_1+R2+1 = 10
R0=0
R1=3
R2=5
L0
L1
L2
0 1 2 3 4 5 6 7 8 9 10 63

Fig. 3. Run-length decoding principle.

Conceptually, for each *run-level* pair returned by the VLD, the run-length decoder outputs the number of zeros specified by the *run* value and then pass the *level* through. In a programmable processor–based platform, a way to optimize this process is to fill in an empty vector with *level* values, L, at positions defined by *run* values, as depicted in Figure 3: the position of a non-zero coefficient, `nz_coeff_pos`, is computed by adding the *run* value, `R`, and an `'1'` to the position of the previous non-zero coefficient. This common strategy has been widely used in previous work [1, 7, 13] and will be used subsequently, too.

In connection with the software implementation of the entropy decoder we propose, we would like to mention that both VLD and RLD are sequential tasks. Consequently, entropy decoding is an intricate function on TriMedia, since a VLIW processor must benefit from instruction-level parallelism in order to be efficient.

The next subsection will outline some elements of the MPEG-2 standard related to variable-length decoding.

2.2 MPEG-2–compliant Variable-Length Decoding

MPEG-2 defines four tables for encoding the DCT coefficients: B12, B13, B14, and B15 [1]. Which table is used depends on the type of image – intra (I) or non-intra (NI), luminance (Y) or chrominance (C) – and a bit-field, `intra_vlc_format`, in the macroblock header, as shown in Table 1. In general, this means that a single stream uses all tables, and the tables can be switched per macroblock and/or block.

`intra_vlc_format`			0	1
I	DC coefficient	Y	B12	B12
		C	B13	B13
	AC coefficient		B14	B15
NI	1st & subsequent coefficient		B14	B14

Table 1. Selection of VLC tables

In the decoding process of DCT coefficients, there are a few exceptional cases to be dealt with:

1. **The DC coefficient** for intra macroblocks: this coefficient is encoded through the B12/B13 tables, depending on the block type: luminance or chrominance.
2. **Escape**: escape code is 6 bits long, followed by 6 bits run and 12 bits signed level.
3. **end-of-block**: this is a 2 or 4 bit code, depending on the `intra_vlc_format` bit.

Apart from these cases, the decoding follows "normal" coding rules. The maximum code-length is 16 bits plus a sign bit. A code determines a *run* and a *level* value. A variable-length code is followed by a sign bit that indicates the sign of the *level* value.

We conclude this section with a review of the TriMedia/CPU64 VLIW core.

2.3 TriMedia/CPU64 architecture

TriMedia/CPU64 is a simulated processor designed to be used in the development process of future 64-bit VLIW cores. Its architecture features a very rich instruction set optimized for media processing. Specifically, TriMedia/CPU64 is a 64-bit 5 issue-slot VLIW core, launching a long instruction every clock cycle [2]. It has a uniform 64-bit wordsize through all functional units, register file, load/store units, on-chip highway and external memory. Each of the five operations in a single VLIW instruction can in principle read two register arguments and write one register result every clock cycle. In addition, each operation can be optionally guarded with the least-significant bit of a fourth register, in order to allow for conditional execution without branch penalty. The architecture supports subword parallelism; for example, operations such as *additions/subtractions*, *shuffle*, *elementwise multiplexing*, on eight 8-bit unsigned integers (`vec64ub`), or on four 16-bit signed integers (`vec64sh`) are possible. Super-operations, which occupy two adjacent slots in the VLIW instruction, and map to a double-width functional unit are also supported. The current organization of the TriMedia/CPU64 core is presented in Figure 4.

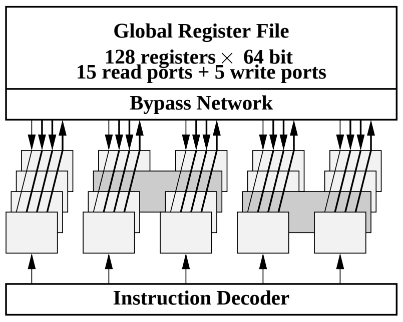


Fig. 4. TriMedia/CPU64 organization.

3 Entropy decoder implementation

According to the reference implementation [7], the VLD is implemented as a repeated table-lookup. Each lookup analyzes a fixed size chunk of bits (for example, `LOOKUP_ADDRESS_WIDTH` = 6 or 8) and determines if a valid code was encountered or some more bits need to be decoded. In any case, the number of consumed bits ranging from the smallest variable-length code to the chunk size is generated. In case of a valid decode, i.e., *hit*, a *run-level* pair is generated, or an *escape* or *end_of_block* flag is set. If a *miss* is detected, i.e., more bits are needed for a valid decode, an offset into the VLC table for a second- or third-level lookup, *table_offset*, is generated. This process of signaling an incomplete decode and generating a new offset may be repeated a number of times, depending on the largest variable-length code and chunk size.

The following basic stages can be discerned in the reference implementation of the entropy decoder on TriMedia/CPU64:

1. **Initializations**.
2. **Barrel-shift the VLC string** according to the *accumulated code-length* value.
3. **Table look-up** (look-up address computation, table look-up proper). The table look-up returns a 32-bit word containing all the fields mentioned at Stage 4.
4. **Field extraction**: *run*, *level*, *code_length*, *valid_decode*, *end_of_block*, *escape*, *table_offset*.
5. **Update (modulo-64) the accumulated code-length**:
   acc_code_length = *acc_code_length* + *code_length*
   If an overflow has been encountered, **advance the VLC string** by 64 bits.
6. **Exit** the loop if *end_of_block* has been encountered.
7. **Handle escape** if *escape* has been encountered.
8. **Run-length decoding**: de-zig-zag, followed by filling-in an empty 8×8 matrix.
9. **Go to** Stage 2.

The Stage 8 – **run-length decoding** – is folded into the loop, such that loop pipelining is employed [7]. That is, the run-length decoding for the previous decoded symbol is carried out simultaneously with the variable-length decoding of the current symbol.

Updating the *acc_code_length* value is carried out modulo-64. The main idea is to match this process with the transfer capabilities of the 64-bit version of TriMedia. That is, a new chunk of 64 bits of information to be decoded is read on overflow. Also, we would like to emphasize that the VLC-related information is stored into the lookup table in a packed format, as 32-bit unsigned integers, as depicted in Table 2. Therefore, a sequence of masking and shifting operations are needed to extract these fields.

Table 2. The original VLC table format.

	end-of-block (stop)	escape	valid	run	level	table offset	code-length
No. of bits	1	1	1	5	8	12	4
Position	31	30	29	28-24	23-16	15-4	3-0

To make the presentation self consistent, the reference implementation of the entropy decoding routine is presented in Algorithm 1. All identifiers written with capital letters are regarded as constants. In the sequel, we will provide some additional information regarding this algorithm, highlighting efficiency-related issues.

Algorithm 1 Entropy decoder routine – reference implementation

```
set-up the test-bench (store the VLC lookup table, read the VLC_string into memory, etc.)

for $i = 1$ to NO_OF_MACROBLOCKS do
  for $j = 1$ to NO_OF_BLOCKS_IN_MACROBLOCK do
    table_offset ← FIRST_TABLE_OFFSET
    nz_coeff_pos_ZZ ← 0
    run ← 0
    valid_decode ← 0

    loop
      barrel-shift the VLC_string with acc_code_length positions
      lookup_address ← the leading LOOKUP_ADDRESS_WIDTH bits from VLC_string
      lookup_address ← lookup_address + table_offset
      retrieved_32_bit_word ← VLC_table[lookup_address]

      nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + run
      nz_coeff_pos ← invZZ_table[nz_coeff_pos_ZZ]
      $8 \times 8$_matrix[nz_coeff_pos] ← level
      nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + valid_decode

      extract code_length, run, level, table_offset, escape, valid_decode, end_of_block from
      retrieved_32_bit_word

      acc_code_length ← acc_code_length + code_length
      if acc_code_length $\leq 64$ and not(escape) then
        continue { ——————————-> go to loop}
      end if
      if end_of_block flag is raised then
        break { ———————————-> initiate the next for iteration (block-level)}
      end if
      if acc_code_length $\geq 64$ then
        advance the VLC_string by 64 bits
        acc_code_length ← acc_code_length - 64
      end if
      if escape flag is raised then
        run ← next 6 bits from VLC_string
        level ← next 12 bits from VLC_string
        acc_code_length ← acc_code_length + 6 + 12
      end if
    end loop
  end for
end for
```

The entropy decoder routine consists of a first **for** loop (lines 3–41) cycling over all macroblocks in the MPEG conformance string, a second **for** loop (lines 4–40) cycling over all blocks in a macroblock, and an inner (infinite) loop labeled **loop** (lines 10–39), cycling over all DCT coefficients in a block. The inner loop is left only when an *end_of_block* is encountered (lines 27–29).

The initializations for block-level decoding are performed at lines 5–8. Table look-up, i.e., variable-length decoding, is carried out at lines 11–13. Lines 15–18 implement run-length decoding, which, as we already mentioned, is folded into the loop in order to employ loop pipelining. Field extraction is performed at line 20. The barrel-shifting (line 11) is done on an 128-bit field, by means of a TriMedia–specific operation:

```
bitfunshift Rsrc_1 Rsrc_2 Rsrc_3 → Rdest_1 Rdest_2
```

where `Rsrc_1` and `Rsrc_2` are the two 64-bit registers storing the leading 128 bits of the VLC_string to be shifted, the `Rsrc_3` defines the shifting value, and `Rdest_1` and `Rdest_2` are the two 64-bit registers storing the 128-bit shifted field. Obviously, only the value stored into `Rdest_1` register will be used for the look-up procedure. It should be mentioned that since *acc_code_length* is updated modulo-64 (lines 30–33), at least 47 bits are available in `Rdest_1` for the next decoding iteration in the most defavorable case (this can be easily verified by assuming that *acc_code_length* = 63 at line 34).

A particular optimization technique has been used in order to keep the most likely iteration (that is when no more incoming bits from the MPEG string are needed, and none of the *escape*, *end_of_block*, and *error* conditions is raised), as short as possible. According to this technique, the *escape* flag is also set to ’1’ when any of the *escape*, *end_of_block*, or *error* conditions occurs. In this way, a jump to the beginning of the inner loop is taken when none of the above mentioned conditions is raised (lines 24–26). All the exceptional cases are managed after this jump: *end_of_block* at lines 27–29, modulo-64 updating and advancing the VLC string at lines 30–33, and *escape* at lines 34–38. It should be mentioned that there is no flag to indicate an *error* condition. When an *error* is encountered, *end_of_block* = 1 and *valid_decode* = 0 simultaneously. Therefore, the loop will be left because the *end_of_block* flag is set. However, it is the responsibility of the entropy decoder calling routine to detect if a valid *end_of_block* has been detected or an *error* has occured. Since this subject is beyond the goal of the paper, it will not be analyzed in the sequel.

In connection to the efficiency of the reference implementation, we would like to specify that the major drawback of the software pipeline is that only variable-length decoding for the first DCT coefficient will be performed during the first iteration, the code associated with run-length decoding being dummy. That is, the method penalty is the overhead needed to fire-up the software pipeline. Since the number of non-zero DCT coefficients in a block is rather small, ranging, for example, between 3.3 and 5.8 for non-intra macroblocks [7], the number of iterations per block is also small. Consequently, this overhead can be significantly large.

In the sequel, we will discuss the improvements that we propose with respect to decoding of non-intra macroblocks. That is, the VLC table will be the B14 table defined by the MPEG standard if we will not state otherwise.

To improve the performance of the entropy decoder, we propose the following changes in respect with the reference implementation:

- **The prologue of the pipelined loop** [14] **is exposed to the compiler**. Since the VLC table does not have "holes" in the region of short code-length coefficients (i.e., each and every entry in the VLC table in that region corresponds either to a short codeword which can be decoded in a single iteration, or to a long codeword which will be decoded in two or more iterations), there are no incoming bit combinations which do not have a meaning within the prologue. Therefore, an *error* condition will never be raised. Moreover, since an *end_of_block* symbol is not allowed for the first coefficient in a block, an *end_of_block* condition will never be encountered, too. Consequently, testing the *end_of_block* flag (lines 27–29 in Algorithm 1) within the prologue becomes superfluous and can be eliminated. For this reason, a very simple code consisting of a first-level look-up, folowed by an extraction of the *code_length, run, level, lookup_address_width, table_offset, escape, valid_decode* (and, notable, no extraction of the *end_of_block* flag) can efficiently fire-up the software pipeline.
- **Barrel-shifting is carried out by means of an extended `bitfunshift` TriMedia specific operations**.

  ```
  bitfunshift_3 Rsrc_1 Rsrc_2 Rsrc_3 Rsrc_4 → Rdest_1 Rdest_2
  ```

  The main idea is to gain flexibility over the modulo-64 operation by performing the barrel-shift operation on $3 \times 64 = 192$ bits instead of $2 \times 64 = 128$ bits. In this way, the modulo-64 operation can be postponed, since additional 64 bits are available for decoding over the standard implementation.
- **The lookup returns a 64-bit value instead of a 32-bit value**. The main idea is to store each of the *code_length*, *run*, *level*, *lookup_address_width* (which defines the chunk size of the next look-up), *table_offset*, *escape*, *valid_decode* (signals a hit), and *end_of_block* fields within the boundaries of a byte (that is, in an unpacked way instead of a packed one). Since extracting a byte from a 64-bit value takes only 1 cycle on TriMedia, our solution is two times faster than using a pair of masking and shifting operations required by the 32-bit approach. The cost of such approach is a double-size look-up table. It is still an open question which approach is better with respect to a particular TriMedia cache size, as the cache misses may become a bottleneck when the performance evaluation is made for a complete MPEG decoder. The new format of the VLC table format is presented in Table 3.
- **The chunk size is variable**, which leads to a more compact look-up table. According to our experiments, there are enough empty slots in the TriMedia instruction format for an entropy decoding task. Consequently, a variable chunk size does not introduce real dependencies.

In connection with the Table 3, several comments should be provided. The VLC table is a one-dimensional array of vectors, where each vector contains eight unsigned bytes. In order to keep the number of instructions as low as possible, we propose to store the sign bit of each and every codeword into the lookup table.

Table 3. The proposed VLC table format.

	code-length	run	level	table offset	lookup address width	escape	valid decode	EOB
No. of bits	8	8	8	8	8	8	8	8
Position	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0

According to Table B14, the *level* value ranges between $-40 \cdots + 40$. Thus, 7 bits (less than 1 byte) are sufficient to represent all the values. However, precautions have to be taken to convert *level* to a signed integer after extraction (Algorithm 2).

Algorithm 2 Converting the *level* from 8-bit unsigned integer to a 16-bit signed integer

```
#define LEVEL_FIELD 5

int16 level;

retrieved_vec64ub = VLC_table[ lookup_address];
level = (int16) ub_get( retrieved_vec64ub, LEVEL_FIELD);
level = (int16)((level ≪ 24) ≫ 24); /* 32-bit processing */
```

The least significant byte has been allocated for *end_of_block* (EOB) flag. Since the TriMedia C compiler recognizes expressions of the form $(E_1\&1)$, the least significant bit of this byte is set to '1' when an *end_of_block* condition is raised. This way, the condition for leaving the loop can be written as follows:

Algorithm 3 TriMedia-specific code for testing the *end-of-block* condition

```
#define END_OF_BLOCK_FIELD 0

uint8 end_of_block;

for (;;) {
  retrieved_vec64ub = VLC_table[ lookup_address];
  end_of_block = ub_get( retrieved_vec64ub, END_OF_BLOCK_FIELD);
  if ( end_of_block & 1)
    break;
}
```

The *table_offset* field defines the partitioning of the B14 into smaller lookup tables. The B14 table has been splitted in eight tables (*first*, *second*, *third*, *forth*, *fifth*, *sixth*, *seventh*, *eighth*) which are presented subsequently. We mention that, in order to improve the readness, we preserved the order of the rows as in the MPEG standard.

VL code	Run	Level
`1s`	0	1
`011s`	1	1
`0100 s`	0	2
`0101 s`	2	1
`0010 1s`	0	3
`0011 1s`	3	1
`0011 0s`	4	1
`0001 10s`	1	2
`0001 11s`	5	1
`0001 01s`	6	1
`0001 00s`	7	1
`0000 110s`	0	4
`0000 100s`	2	2
`0000 111s`	8	1
`0000 101s`	9	1
`0000 01`	Escape	

Table 4. First table

VL code	Run	Level
`10`	End of Block	
`11s`	0	1
`011s`	1	1
`0100 s`	0	2
`0101 s`	2	1
`0010 1s`	0	3
`0011 1s`	3	1
`0011 0s`	4	1
`0001 10s`	1	2
`0001 11s`	5	1
`0001 01s`	6	1
`0001 00s`	7	1
`0000 110s`	0	4
`0000 100s`	2	2
`0000 111s`	8	1
`0000 101s`	9	1
`0000 01`	Escape	

Table 5. Second table

1st prefix	VL code	Run	Level
`0010 0`	`110 s`	0	5
	`001 s`	0	6
	`101 s`	1	3
	`100 s`	3	2
	`111 s`	10	1
	`011 s`	11	1
	`010 s`	12	1
	`000 s`	13	1

Table 6. Third table

1st prefix	VL code	Run	Level
`0000 001`	`0 10s`	0	7
	`1 00s`	1	4
	`0 11s`	2	3
	`1 11s`	4	2
	`0 01s`	5	2
	`1 10s`	14	1
	`1 01s`	15	1
	`0 00s`	16	1

Table 7. Forth table

1st prefix	VL code	Run	Level
`0000 0001`	`1101 s`	0	8
	`1000 s`	0	9
	`0011 s`	0	10
	`0000 s`	0	11
	`1011 s`	1	5
	`0100 s`	2	4
	`1100 s`	3	3
	`0010 s`	4	3
	`1110 s`	6	2
	`0101 s`	7	2
	`0001 s`	8	2
	`1111 s`	17	1
	`1010 s`	18	1
	`1001 s`	19	1
	`0111 s`	20	1
	`0110 s`	21	1

Table 8. Fifth table

1st prefix	VL code	Run	Level
`0000 0000`	`1101 0s`	0	12
	`1100 1s`	0	13
	`1100 0s`	0	14
	`1011 1s`	0	15
	`1011 0s`	1	6
	`1010 1s`	1	7
	`1010 0s`	2	5
	`1001 1s`	3	4
	`1001 0s`	5	3
	`1000 1s`	9	2
	`1000 0s`	10	2
	`1111 1s`	22	1
	`1111 0s`	23	1
	`1110 1s`	24	1
	`1110 0s`	25	1
	`1101 1s`	26	1
	`0111 11s`	0	16
	`0111 10s`	0	17
	`0111 01s`	0	18
	`0111 00s`	0	19
	`0110 11s`	0	20
	`0110 10s`	0	21
	`0110 01s`	0	22
	`0110 00s`	0	23
	`0101 11s`	0	24
	`0101 10s`	0	25
	`0101 01s`	0	26
	`0101 00s`	0	27
	`0100 11s`	0	28
	`0100 10s`	0	29
	`0100 01s`	0	30
	`0100 00s`	0	31

Table 9. Sixth table

1st prefix	2nd prefix	VL code	Run	Level
`0000 0000`	`001`	`1 000s`	0	32
		`0 111s`	0	33
		`0 110s`	0	34
		`0 101s`	0	35
		`0 100s`	0	36
		`0 011s`	0	37
		`0 010s`	0	38
		`0 001s`	0	39
		`0 000s`	0	40
		`1 111s`	1	8
		`1 110s`	1	9
		`1 101s`	1	10
		`1 100s`	1	11
		`1 011s`	1	12
		`1 010s`	1	13
		`1 001s`	1	14

Table 10. Seventh table

1st prefix	2nd prefix	VL code	Run	Level
`0000 0000`	`0001`	`0011 s`	1	15
		`0010 s`	1	16
		`0001 s`	1	17
		`0000 s`	1	18
		`0100 s`	6	3
		`1010 s`	11	2
		`1001 s`	12	2
		`1000 s`	13	2
		`0111 s`	14	2
		`0110 s`	15	2
		`0101 s`	16	2
		`1111 s`	27	1
		`1110 s`	28	1
		`1101 s`	29	1
		`1100 s`	30	1
		`1011 s`	31	1

Table 11. Eighth table

All eight tables are stored into memory one after another, i.e., in a concatenated way. The number of address bits for each table is related to the maximum length of the variable-length codes. That is, Tables *first* and *second* have each 8 address bits, Table *sixth* has 7 address bits, Tables *third* and *forth* have each 4 address bits, and Tables *fifth*, *seventh*, and *eighth* have each 5 address bits. Thus, the sizes of the tables are as follows:

Table	No. of address lines (*lookup_address_width*)	Size (64-bit words)	*table_offset*
first	8	$2^8 =$ 256	`0`
second	8	$2^8 =$ 256	`0x100`
third	4	$2^4 =$ 16	`0x200`
forth	4	$2^4 =$ 16	`0x210`
fifth	5	$2^5 =$ 32	`0x220`
sixth	7	$2^7 =$ 128	`0x240`
seventh	5	$2^5 =$ 32	`0x2c0`
eighth	5	$2^5 =$ 32	`0x2e0`

Table 12. Number of address lines, size, and offset for each VLC table

with a total of 768 64-bit words, which means 6 KB.

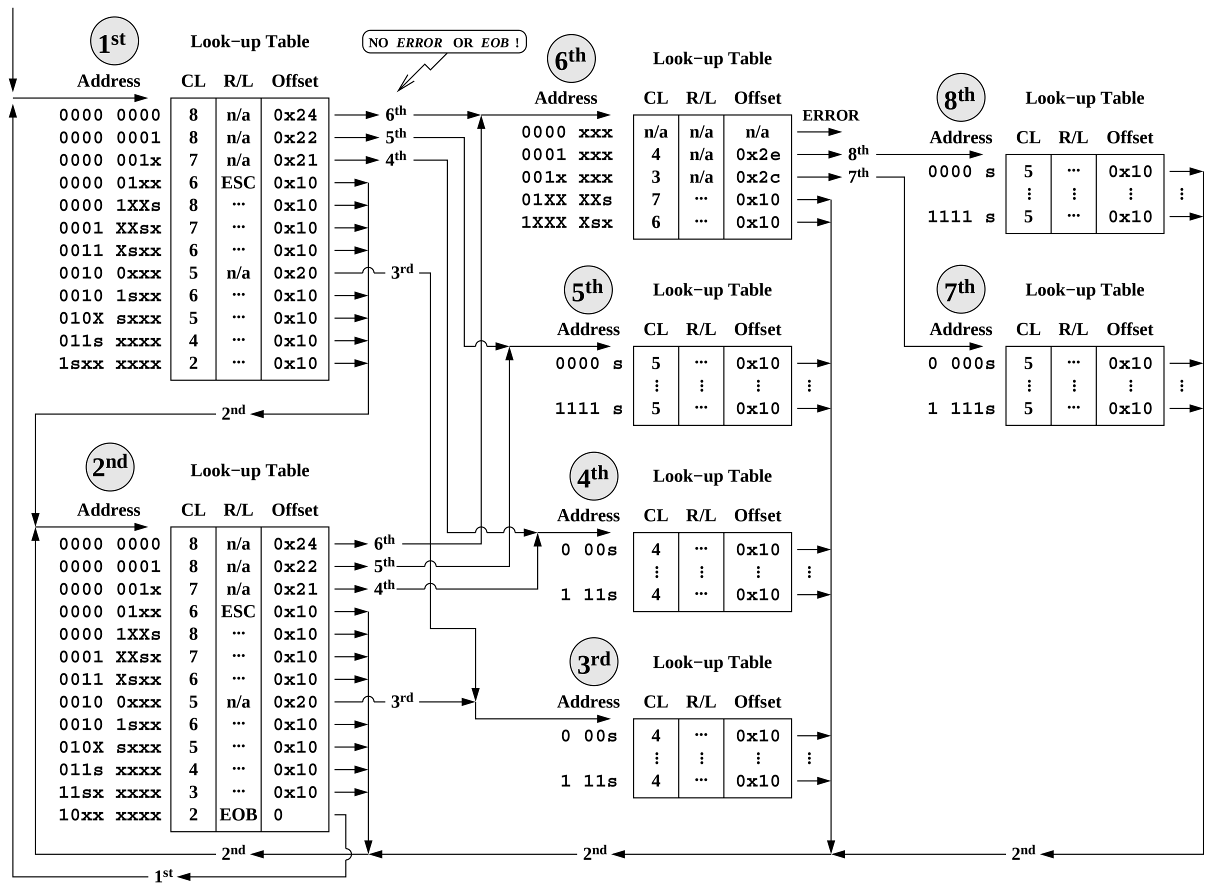
The decoding procedure can be exemplified on Figure 5. Let us suppose that the following string is to be decoded: `1000000000011000110....` The *table_offset* is initialized to `0`, that is the *first* table is being pointed to. Also, *lookup_address_width* is initialized to `8`, which means that the first 8 bits of the string, i.e., `10000000`, will be regarded as address into the *first* table. The following values are retrieved: *code_length* = 2, *run* = 0, *level* = 1, *table_offset* = `0x100`, and *lookup_address_width* = `8`. Which means that the *second* table will be accessed during the second iteration.

After shifting out the two bits decoded at the previous iteration, the leading eight bits, i.e., *00000000*, will be regarded as address, this time into the *second* table. By looking-up, *code_length* = `8`, *table_offset* = `0x240`, and *lookup_address_width* = `7`. That is, the *sixth* table will be accessed. No valid *run-level* pair has been detected.

At this moment the *accumulated_code_length* is 10. Therefore, the leading 10 bits have to be shifted out from the input string. Then, the next seven bits, i.e., *0011000*, are regarded as address into the *sixth* table. Again, no valid *run-level* pair is detected. The *code_length* = `3`, *table_offset* = `0x2c0`, *lookup_address_width* = `5`. That is, the *seventh* table will be accessed.

After incrementation, the *accumulated-code-length* = 13. After shifting out the leading 13 bits, the next five bits, i.e., *10001* are the address into the *seventh* table. The look-up procedure retrieves the following values: *code_length* = 5, *run* = `0`, *level* = `-32`, *lookup_address_width* = `8`, *table_offset* = `0x100` bypassing the *first* table. That is, all subsequent coefficients of the 8×8 block will use only the Tables *second - eighth.*

Finally, the accumulated-code-length is 18. The next eight bits to be sent as address to the *second* table are: *10xxxxxx.* An *end_of_block* symbol is detected, and the *table-offset = 0*; that is, the *first* table is to be accessed for decoding of a new block.



1st Look-up Table

Address	CL	R/L	Offset
0000 0000	8	n/a	0x24
0000 0001	8	n/a	0x22
0000 001x	7	n/a	0x21
0000 01xx	6	ESC	0x10
0000 1XXs	8	···	0x10
0001 XXsx	7	···	0x10
0011 Xsxx	6	···	0x10
0010 0xxx	5	n/a	0x20
0010 1sxx	6	···	0x10
010X sxxx	5	···	0x10
011s xxxx	4	···	0x10
1sxx xxxx	2	···	0x10

2nd Look-up Table

Address	CL	R/L	Offset
0000 0000	8	n/a	0x24
0000 0001	8	n/a	0x22
0000 001x	7	n/a	0x21
0000 01xx	6	ESC	0x10
0000 1XXs	8	···	0x10
0001 XXsx	7	···	0x10
0011 Xsxx	6	···	0x10
0010 0xxx	5	n/a	0x20
0010 1sxx	6	···	0x10
010X sxxx	5	···	0x10
011s xxxx	4	···	0x10
11sx xxxx	3	···	0x10
10xx xxxx	2	EOB	0

6th Look-up Table

Address	CL	R/L	Offset
0000 xxx	n/a	n/a	n/a
0001 xxx	4	n/a	0x2e
001x xxx	3	n/a	0x2c
01XX XXs	7	···	0x10
1XXX Xsx	6	···	0x10

5th Look-up Table

Address	CL	R/L	Offset
0000 s	5	···	0x10
⋮	⋮	⋮	⋮
1111 s	5	···	0x10

4th Look-up Table

Address	CL	R/L	Offset
0 00s	4	···	0x10
⋮	⋮	⋮	⋮
1 11s	4	···	0x10

3rd Look-up Table

Address	CL	R/L	Offset
0 00s	4	···	0x10
⋮	⋮	⋮	⋮
1 11s	4	···	0x10

8th Look-up Table

Address	CL	R/L	Offset
0000 s	5	···	0x10
⋮	⋮	⋮	⋮
1111 s	5	···	0x10

7th Look-up Table

Address	CL	R/L	Offset
0 000s	5	···	0x10
⋮	⋮	⋮	⋮
1 111s	5	···	0x10

Fig. 5. The flowchart of the variable-length decoding procedure.

The entropy decoder implementation we propose is presented in Algorithm 4. As it can be observed, the prologue of the inner (infinite) loop (lines 17–45) has been exposed to the compiler (lines 4–15). Since an *end_of_block* or *error* condition will never occur on the first table lookup (line 7), testing the *end_of_block* condition during the prologue becomes superfluous and, therefore, has been eliminated.

Special considerations have to be provided with respect to modulo-64 operation. As me already mentioned, since the extended `bitfunshift` TriMedia-specific operation is used, more flexibility in postponing the modulo-64 operation is gained. Indeed, there is no such operation within the prologue. However, from the MPEG syntax point of view this is not entirely correct. Assuming that *acc_code_length* is 63 at line 36, it will become 81 at line 45. Considering that an *end_of_block* is encountered, then *acc_code_length* = 83. If this situation occurs during the decoding of the first block in a macroblock, and if the subsequent five coded blocks in the same macroblock include each an *escape* sequence followed by an *end_of_block*, then *acc_code_length* $= 83 + 5 \times 24 + 5 \times 2 = 213$, that is more than the limit of 192 bits. Fortunately, this case is not statistically relevant (we did verify it on all MPEG conformance strings mentioned in the subsequent section). Fortunately, this exceptional situation can be overcomed without much penalty by augmenting the *escape* handling code within the prologue (lines 11–15) with a modulo-64 operation.

The same strategy of exposing the prologue of the loop to the compiler can be applied for decoding of intra blocks, since an *end_of_block* can never occur during the decoding of an DC coefficient. However, special precautions have to be taken in order to deal with errors.

Finally, it should be mentioned that standard optimization techniques such as *loop unrolling* or *grafting* [15] cannot be applied, because that would introduce awkward *escape* code and/or barrel-shifting processing.

4 Experimental results

The testing database for our entropy decoder consists of a number of pre-processed MPEG conformance strings, or *scenes*, from which all the data not representing DCT coefficients have been removed. Therefore, such strings include only *run-level* and *end-of-block* symbols.

For all experiments described subsequently, the MPEG-compliant bit string is assumed to be entirely resident into the main memory. In this way, side effects associated with bit string acquisition such as asynchronous interrupts, trashing routines, or other operating system related tasks, do not have to be counted. Moreover, saving the reconstructed 8×8 matrices into memory, as well as zeroing these matrices in order to initialize a new entropy decoding task are equally not considered. Since both procedures can be considered parts of adjacent tasks, such as IDCT or motion compensation, they are subject to further optimizations at the complete MPEG decoder level. Thus, in our experiments, the run-length decoder will overwrite the same 8×8 matrices again and again. With these assumptions, the only relevant metric is the number of instruction cycles required to perform strictly entropy decoding. Therefore, the main goal was to minimize this number.

Algorithm 4 Entropy decoder routine with the prologue exposed to the compiler

```
1: for $i = 1$ to NO_OF_MACROBLOCKS do
2:   for $j = 1$ to NO_OF_BLOCKS_IN_MACROBLOCK do
3:     nz_coeff_pos_ZZ ← 0
4:     barrel-shifting the VLC_string
5:     lookup_address ← the leading FIRST_LOOKUP_ADDRESS_WIDTH bits from VLC_string
6:     lookup_address ← lookup_address + (FIRST_TABLE_OFFSET $\ll$ 4)
7:     retrieved_vec64ub ← VLC_table[lookup_address]
8:
9:     extract code_length, run, level, table_offset, lookup_address_width, escape,
       valid_decode from retrieved_vec64ub {end_of_block field is not extracted!}
10:    acc_code_length ← acc_code_length + code_length
11:    if escape flag is raised then
12:      run ← next 6 bits from VLC_string
13:      level ← next 12 bits from VLC_string
14:      acc_code_length ← acc_code_length + 6 + 12
15:    end if
16:
17:    loop
18:      barrel-shift the VLC_string
19:      lookup_address ← the leading lookup_address_width bits from VLC_string
20:      lookup_address ← lookup_address + table_offset
21:      retrieved_vec64ub ← VLC_table[lookup_address]
22:
23:      nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + Run
24:      nz_coeff_pos ← invZZ_table[nz_coeff_pos_ZZ]
25:      8 × 8_matrix[nz_coeff_pos] ← Level
26:      nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + valid_decode
27:
28:      extract code_length, run, level, table_offset, lookup_address_width, escape,
         valid_decode, end_of_block from retrieved_vec64ub
29:      acc_code_length ← acc_code_length + code_length
30:      if acc_code_length $\leq 64$ and not(escape) then
31:        continue { ——————————> go to loop}
32:      end if
33:      if end_of_block flag is raised then
34:        break { ————————————> initiate the next for iteration (block-level)}
35:      end if
36:      if acc_code_length $\geq 64$ then
37:        advance the VLC_string by 64 bits
38:        acc_code_length ← acc_code_length - 64
39:      end if
40:      if escape flag is raised then
41:        run ← next 6 bits from VLC_string
42:        level ← next 12 bits from VLC_string
43:        acc_code_length ← acc_code_length + 6 + 12
44:      end if
45:    end loop
46:  end for
47: end for
```

Table 13. **Entropy decoding experimental results.**

Scene (*.m2v)	Block type	Workload (coeff.)	Scene profile (bit/coeff.)	Reference implementation (cycle/coeff.)	Proposed implementation (cycles)	Proposed implementation (cycle/coeff.)	**Improvement**
batman	I (B15)	172,745	5.5	21.85	2,843,376	16.5	22.5 %
	NI	266,485			4,592,358	17.2	
popplen	I (B15)	47,003	5.3	20.19	777,553	16.5	17.3 %
	NI	28,069			475,326	16.9	
sarnoff2	I (B14)	80,563	5.1	21.9	1,387,489	17.2	23.3 %
	NI	36,408			577,388	15.9	
tennis	I (B14)	12,345	6.1	21.77	210,011	17.0	20.7 %
	I (B15)	120,754			1,937,808	16.0	
	NI	137,756			2,527,395	18.3	
ti1cheer	I (B15)	80,818	5.1	20.75	1,311,687	16.2	21.9 %
	NI	51,680			836,082	16.2	

The results for entropy decoder are presented in Table 13. The figures indicate the number of instruction cycles needed to decode the pre-processed MPEG string. The last column of the table specifies the relative improvement of the proposed entropy decoder versus its reference counterpart. Unfortunately, only the average number of cycles per coefficient has been disclosed for the reference implementation [7].

It is also worth mentioning that the absolute performance of the proposed entropy decoder ranges between $15.9 \div 18.3$ cycles/coeff., with the mean 16.9 cycles/coeff. This is a very good result with respect to both 33.0 cycles/coeff. needed for variable-length decoding and Inverse Quantization (IQ) on a Pentium processor with MultiMedia eXtension (MMX) claimed by Ishii et. al [8], and 26.0 cycles/coeff. achieved on an TMS320C80 media video processor by Bonomini *et al.* [9]. The additional IQ functionality considered by the referred papers is not a real concern for us, since our preliminary results indicate that a significant number of operations related to inverse quantization can be still scheduled in the delay slots of the table lookup.

To make an absolute estimation of the performance we achieved, we mention that the maximum MPEG-2 compressed bit rate for Main Profile – Main Level (MP@ML) is 15 Mbit/s. For 16.9 cycle/coefficient, and an average of 5.4 bit/coefficient [7], the following rate can be processed in real-time by our implementation:

$$5.4\,\frac{\text{bit}}{\text{coefficient}} \;\times\; 200\cdot 10^6\,\frac{\text{cycle}}{\text{sec}} \;\times\; \frac{1}{16.9}\,\frac{\text{coefficient}}{\text{cycle}} \quad\approx\quad 64\,\frac{\text{M}bit}{\text{sec}}$$

That means that less than one-quarter of the computing power of the processor is used, or, equivalently, four MP@ML strings can be simultaneously (entropy) decoded.

5 Conclusions

We proposed a new entropy decoder implementation on TriMedia/CPU64 processor VLIW core which has the prologue exposed to the compiler. The VLC tables are organized in a special way such that an *end_of_block* or *error* will never be encountered during the prologue. By running preprocessed MPEG-2 conformance strings including only *run-level* and *end_of_block* symbols, we determined that the proposed entropy decoder is approximately 20% faster than its reference counterpart. In future work, we intend to evaluate the performance improvement for a complete MPEG decoder.

References

1. ***: MPEG-2 Video Codec. MPEG Software Simulation Group, WWW address: http://www.mpeg.org/MPEG/MSSG/
2. van Eijndhoven, J.T.J., Sijstermans, F.W., Vissers, K.A., Pol, E.J.D., Tromp, M.J.A., Struik, P., Bloks, R.H.J., van der Wolf, P., Pimentel, A.D., Vranken, H.P.E.: TriMedia CPU64 Architecture. In: IEEE Proceedings of International Conference on Computer Design (ICCD 1999), Austin, Texas (1999), 586–592.
3. ***: TM-1000 Data Book. Philips Electronics North America Corporation, TriMedia Product Group, Sunnyvale, California (1998).
4. Riemens, A.K., Vissers, K.A., Schutten, R.J., Sijstermans, F.W., Hekstra, G.J., Hei, G.D.L.: TriMedia CPU64 Application Domain and Benchmark Suite. In: IEEE Proceedings of International Conference on Computer Design (ICCD 1999), Austin, Texas (1999), 580–585.
5. Sun, M.T.: Design of High-Throughput Entropy Codec. In: VLSI Implementations for Image Communications. Volume 2. Elsevier Science Publishers B.V., Amsterdam, The Netherlands (1993), 345–364.
6. Mitchell, J.L., Pennebaker, W.B., Fogg, C.E., LeGall, D.J.: MPEG Video Compression Standard. Chapman & Hall, New York, New York (1996).
7. Pol, E.J.D.: VLD Performance on TriMedia/CPU64. Private Communication (2000).
8. Ishii, D., Ikekawa, M., Kuroda, I.: Parallel Variable Length Decoding with Inverse Quantization for Software MPEG-2 Decoders. In: Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS97), Leicester, United Kingdom (1997), 500–509.
9. Bonomini, F., Marco-Zompit, F.D., Milan, G., Odorico, A., Palumbo, D.: Implementing an MPEG2 Video Decoder Based on the TMS320C80 MVP. Application Report SPRA332, Texas Instruments, Paris, France (1996).
10. Haskell, B.G., Puri, A., Netravali, A.N.: Digital Video: An Introduction to MPEG-2. Kluwer Academic Publishers, Norwell, Massachusetts (1996).
11. Lei, S.M., Sun, M.T.: An Entropy Coding System for Digital HDTV Applications. In: IEEE Transactions on Circuits and Systems for Video Technology **1** (1991), 147–155.
12. Lin, H.D., Messerschmitt, D.G.: Finite State Machine has Unlimited Concurrency. In: IEEE Transactions on Circuits and Systems **38** (1991), 465–475.
13. Sima, M., Cotofana, S., Vassiliadis, S., van Eijndhoven, J.T.J., Vissers, K.: MPEG Macroblock Parsing and Pel Reconstruction on an FPGA-augmented TriMedia Processor. In: IEEE Proceedings of International Conference on Computer Design (ICCD 2001), Austin, Texas (2001), 425–430.
14. Johnson, W.M. *Superscalar Microprocessor Design,* Prentice Hall, Englewood Cliffs, New Jersey (1991).
15. ***: Book 2 – Cookbook. Part D: Optimizing TriMedia Applications. TriMedia Technologies, Inc., TriMedia Technologies, Inc., Milpitas, California (2000).

Stride Permutation Access in Interleaved Memory Systems

Jarmo Takala and Tuomas Järvinen

Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland
{jarmo.takala, tuomas.jarvinen}@tut.fi

Abstract. In this paper, a conflict-free parallel access scheme for stride permutation access on matched interleaved memory systems is proposed. It is assumed that the number of parallel independent memory modules and the length of array to be accessed are powers-of-two. The proposed scheme supports all the power-of-two strides from 1 to $N/2$ where N is the array length. Structure of the corresponding address generator is also discussed.

1 Introduction

Over the years several techniques have been proposed to increase data transfer rates between memory and computational resources in processor architectures. One well-established technique for such a purpose is memory interleaving where data is distributed over multiple independent memory banks or modules. In general, the interleaved memory systems exploit either time or space multiplexing.

Time-multiplexed memories are used to match the processor cycle time and memory access time. Memory accesses will require t cycles to complete and this delay is hidden by sending t access requests to the memory system over a single bus at consecutive cycles. Each request is sent into a different memory bank. If the operands lie in the same memory bank, the second access can be performed after t cycles. Principal block diagram of a time-multiplexed memory is illustrated in Fig. 1(a).

Space-multiplexed memories are used in SIMD processing, i.e., several access requests are sent to the memory over multiple buses, thus the memory latency is not hidden. The memory system requires an interconnection network for providing communication path from processing units to different memory banks. Principal block diagram of a space-multiplexed memory is shown in Fig. 1(b).

In both the previous systems, the memory bandwidth is increased by allowing several simultaneous memory accesses to be directed to different memory modules. If Q accesses can be distributed over Q modules in such a way that all the modules are referenced, Q-fold speedup can be achieved.

Unfortunately the operands to be accessed in parallel often lie in the same memory module thus the parallel access can not be performed. Such a situation is referred to as a conflict and results in a substantial performance degradation. Therefore, the principal problem in interleaved memory systems is to find a

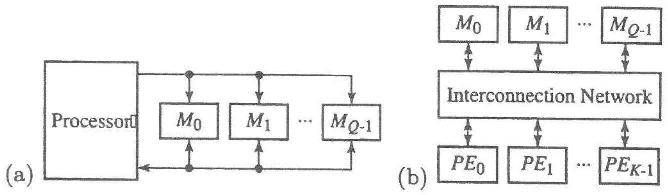


Fig. 1. Interleaved memory systems: (a) time-multiplexed and (b) space-multiplexed. M_k: Memory module. PE_k: Processing element.

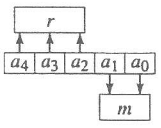
method to distribute data over the memory modules in such a way that conflicts are avoided. For this research problem several solutions have been proposed, which assume that parallel accesses are most likely to be made to subsections of matrices such as rows, columns, or diagonals. The developed methods try to support as many access patterns as possible or provide conflict-free access to specific access patterns.

The method distributing data over modules is referred to as an access scheme, which is a function mapping addresses into storage locations. Since the memory system contains Q memory modules, a storage scheme performs two mappings; it maps an $N = 2^n$-bit address $a = (a_{n-1}, a_{n-2}, \ldots, a_0)^T$ into a $\lceil \log Q \rceil$-bit module address m and into a row address r defining the storage location in the selected memory module.

The most simple access scheme is to obtain row and module addresses by extracting fields from the address a, i.e., $m = a \bmod Q; r = \lfloor a/Q \rfloor$. Such a scheme, low order interleaving, is illustrated in Fig. 2(a). This scheme performs well in linear access but the performance is degraded when other type of access patterns are used [1]. Row-rotation (alternatively skewed) scheme was introduced in [2] to support larger set of access patterns. In principle, the row address r is generated as earlier but the module address is formed by extracting two $\log Q$-bit fields from the address a. The two fields are added to obtain the module address, $m = (a \bmod Q) + (\lfloor a/Q \rfloor \bmod Q)$ as shown in Fig. 2(b). In [3], row-rotation scheme was generalized as a periodic storage scheme, which supports irregular and overlapped access patterns. Several improvements for this scheme has been proposed; in [4], a scheme supporting rows, columns, diagonals, coils, band diagonals, block diagonals, etc. is proposed.

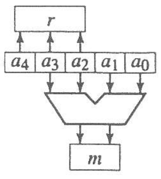
In [5], the adder used in module address generation in row-skewing schemes was replaced by bit-wise exclusive-OR (XOR) operation as depicted in Fig. 2(c). Such a scheme is a linear transformation and sometimes called as a XOR scheme. The linear transformations have two advantages over row-rotation schemes: computation of module address is independent on the number of memory modules and it has flexibility in performing address mappings [1]. The flexibility of the interleaved memory system is even improved if the number of memory modules is larger than the number of parallel accesses. Especially prime numbers of modules are powerful [6].

(a)



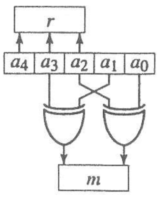
m	**0**	**1**	**2**	**3**
	0	1	2	3
	4	5	6	7
	8	9	10	11
	12	13	14	15
	16	17	18	19
	20	21	22	23
	24	25	26	28
	28	29	30	31

(b)



m	**0**	**1**	**2**	**3**
	0	1	2	3
	7	4	5	6
	10	11	8	9
	13	14	15	12
	16	17	18	19
	23	20	21	22
	26	27	24	25
	29	30	31	28

(c)



m	**0**	**1**	**2**	**3**
	0	1	2	3
	5	4	7	6
	10	11	8	9
	15	14	13	12
	16	17	18	19
	21	20	23	22
	26	27	24	25
	31	30	29	28

Fig. 2. Examples of access schemes for a 32-element vector on a 4-module system: (a) low-order interleaving, (b) row-rotation, and (c) linear transformation.

One specific, often used access pattern is stride access where the consecutive accesses differ by a constant amount k, i.e., every kth element is accessed. In [7], a linear transformation for matched systems, i.e., the number of parallel accesses is the same as the number of modules, is proposed, which supports several power-of-two strides but the implementation is complex, especially if several array lengths need to be supported.

A linear transformation scheme supporting a single stride but several array lengths is proposed in [8]. Conflict-free access of strides can be performed for any array length and any initial address. The implementation is extremely simple requiring only bit-wise XOR operations and a shifter for address field extraction. Support for several strides is considered in [1] but the number of memory modules needs to be greater than the number of parallel accesses. However, the scheme supports strides of form $a2^s$. In [9], linear transformations has been discussed both in matched and unmatched systems.

While the memory interleaving has been studied in supercomputer area, it has received a little consideration in embedded systems. In [10], a memory synthesis method supporting interleaved memories is proposed. The address generation is based on look-up tables containing the access vectors present in the given application. In [11], a method is proposed for reducing the number of conflicts in given application and reordering the accesses such that the number of memory modules is minimized. An alternative approach is taken in [12], where the objective is to reschedule the order of accesses such that the data can be distributed over several memories.

In this paper, a linear transformation scheme for a specific access pattern, stride permutation access, is proposed. The scheme provides conflict-free access to all the strides of powers-of-two for an array of a power-of-two length. The address computation consists of bit-wise XOR operations and results in less complex implementation than the previously reported schemes supporting several strides.

The organization of the paper is the following. In Section 2, stride permutation is defined and some of its properties are given. Motivation for developing an access scheme for this access pattern is provided by describing some applications exploiting stride permutations. In Section 3, constraints for stride access schemes are given. The proposed access scheme is described in detail and its implementation is discussed. Conclusions are provided in Section 4.

2 Stride Permutation

Stride permutations can be described with the aid of matrix transpose; stride-by-S permutation of an N-element vector can be performed by dividing the vector into S-element sub vectors, organizing them into $S \times (N/S)$ matrix form, transposing the obtained matrix, and rearranging the result back to the vector representation [13]. This interpretation implies that the stride S has to be a factor of vector length, i.e., N rem $S = 0$ where rem denotes remainder after division. Another interpretation is to use indexing functions as used in the following formal definition.

Definition 1 (Stride Permutation). *Let us assume a vector $X = (x_0, x_1, \ldots, x_{N-1})$. Stride-by-$S$ permutation reorders X as $Y = (x_{f_{N,S}(0)}, x_{f_{N,S}(1)}, \ldots, x_{f_{N,S}(N-1)})^T$ where the index function $f_{N,S}(i)$ is given as*

$$f_{N,S}(i) = (iS \bmod N) + \lfloor iS/N \rfloor \mid N \text{ rem } S = 0,\ i = 0, 1, \ldots, N-1 \qquad (1)$$

where $\lfloor \cdot \rfloor$ is the floor function.

The stride permutation can also be expressed in matrix form as $Y = P_{N,S}X$ where $P_{N,S}$ is stride-by-S permutation matrix of order N defined as

$$[P_{N,S}]_{mn} = \begin{cases} 1, \text{ iff } n = (mS \bmod N) + \lfloor mS/N \rfloor \\ 0, \text{ otherwise} \end{cases}, m, n = 0, 1, \ldots, N-1 \quad (2)$$

For example, the permutation matrix $P_{8,2}$ associated to stride-by-2 permutation of an 8-element vector is the following (blank entries represent zeroes):

$$P_{8,2} = \begin{pmatrix} 1 & & & & & & & \\ & & 1 & & & & & \\ & & & & 1 & & & \\ & & & & & & 1 & \\ & 1 & & & & & & \\ & & & 1 & & & & \\ & & & & & 1 & & \\ & & & & & & & 1 \end{pmatrix}.$$

In this paper, we limit ourselves to practical cases where the stride and array lengths are powers of two, $N = 2^n, S = 2^s$. Some properties of stride permutations in such cases are provided in the following.

Theorem 1 (Factorization of stride permutations). *Let $ab \leq N$, then*

$$P_{N,ab} = P_{N,a}P_{N,b} = P_{N,b}P_{N,a} \tag{3}$$

The proof for the previous theorem can be found, e.g., from [14].

Corollary 1 (Periodicity). *Stride permutations are periodic with the following properties.*

1) Period of $P_{2^n,2^s}$ is $\operatorname{lcm}(n,s)/s$ where $\operatorname{lcm}(a,b)$ denotes the least common multiple of n and s. In other words,

$$I_{2^n} = \prod_{1}^{\operatorname{lcm}(n,s)/s} P_{2^n,2^s}. \tag{4}$$

2) Consecutive stride permutations always result in a stride permutation:

$$P_{2^n,2^a}P_{2^n,2^b} = P_{2^n,2^{(a+b) \bmod n}}. \tag{5}$$

Proof. 2) If $a + b > n$, the product in (5) can be written as $P_{2^n,2^{kn+(a+b) \bmod n}} = P_{2^n,2^{kn}}P_{2^n,2^{(a+b) \bmod n}}$ where $k > 1$ is an integer. By substituting 2^n for S in (1), we find that $P_{2^n,2^n} = P_{2^n,1} = I_{2^n}$. Therefore, $P_{2^n,2^{kn}} = I_{2^n}$ and the result follows.

1) Let us assume that period of $P_{2^n,2^s}$ is k, thus $ks \bmod n = 0$, i.e., ks is a multiple of n. This implies that k is a multiple of n/s, i.e., $k = mn/s$. k has to be integer, thus s has to be a factor of mn. The smallest number fulfilling the requirement is $\operatorname{lcm}(n,s)$ and, therefore, $k = \operatorname{lcm}(n,s)/s$.

Stride permutations have several practical applications. For example, the previously discussed matrix transpose interpretation of stride permutation implies that an $N \times N$ matrix in a vector form can be transposed by reordering the N^2-element vector according to stride-by-N permutation. Therefore, $N \times N$

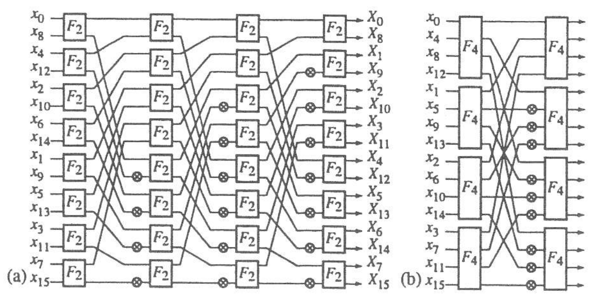


Fig. 3. Signal flow graphs of FFT algorithms: (a) radix-2 and (b) radix-4 algorithm. F_k: k-point FFT.

matrix stored into a memory array can be transposed by accessing its elements in stride-by-N order.

The well known perfect shuffle permutation is a special case of stride permutation, namely stride-by-$N/2$ permutation of an N-point sequence, $P_{N,N/2}$. Perfect shuffle has close relation to several practical algorithms; e.g., Cooley-Tukey radix-2 fast Fourier transform (FFT) algorithm can be scheduled into a form where the interconnections between the processing columns of the signal flow graph are perfect shuffles. Radix-2 algorithms can also be derived into a form where the topology is according to stride-by-2 permutation as illustrated in Fig. 3(a). In radix-4 FFT algorithms, the interconnections can be stride-by-4 permutations as depicted in Fig. 3(b). Fast algorithms for other discrete trigonometric transforms with corresponding topology exist, e.g., for discrete sine, cosine, and Hartley transforms [15, 16].

Stride permutations can be found also in trellis coding and especially in Viterbi algorithm used for decoding of convolutional codes. Convolutional encoders are often described with the aid of a shift register model illustrated in Fig. 4. The state of the encoder X_t at a given time instant t is defined by the contents of the shift register. In $1/n$-rate codes, a single bit is fed into the shift register at a time, thus there are two possible state transitions. This results in a trellis diagram where the transitions form perfect shuffle as depicted in Fig. 4(a). In $2/n$-rate codes, two bits enter the shift register at a time, thus four state transitions are possible, which results in a stride-by-4 permutation in the interconnection as shown in Fig. 4(b).

The previous examples show that stride permutations have practical and important applications in digital signal processing and telecommunications. Applications in these areas are often hard real-time constrained and realized in systems with relatively low clock frequencies, e.g., for extending battery life. Therefore, parallel implementations are preferred, which implies also need to access several operands simultaneously to increase the memory bandwidth.

Typical realizations for all the previous applications are recursive, i.e., small kernels operate over an data array and the results of an iteration are used as op-

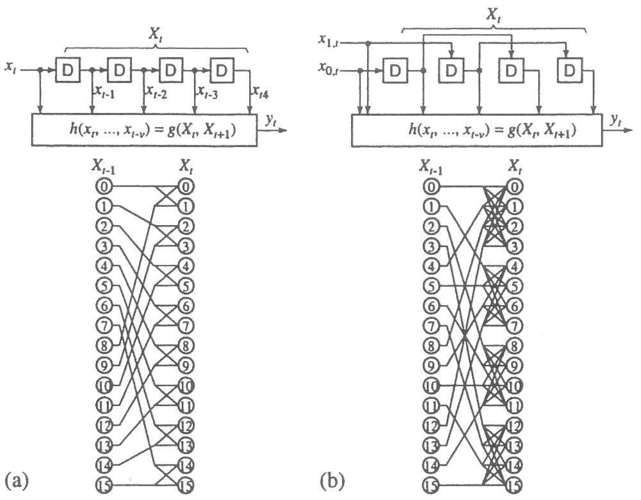


Fig. 4. Single shift register convolutional encoders and allowed state transitions: (a) $1/n$-rate code and (b) $2/n$-rate code. x_t: input at time instant t. X_t: state at time instant t. y_t: output at time instant t. D: bit register.

erands in the next iteration. In addition, the read is performed in different order than write; e.g., in Viterbi decoding of code illustrated in Fig. 4(a), operands are read in perfect shuffle order ($P_{16,8}$) and the results are stored in linear order ($P_{16,1}$). If a single memory array is used, the results should be stored into the same memory locations where the operands were obtained but now the results will be in perfect shuffle order $P_{N,N/2}$, not in linear order, $P_{N,1}$, as intended originally. Therefore, the next $P_{N,N/2}$ accesses should be performed in $P_{N,N/2}$ order. According to Corollary 1, the next read access should be performed in $P_{N,N/2}$ order to compensate the previous additional reordering. Respectively, the next read should be according to $P_{N,N/8}$. Eventually we find that $\log N$ different strides are needed, i.e., all the strides from 1 to $N/2$.

The need to develop an access mode for stride permutations can be illustrated with an example by referring to Fig. 2. In this example, a 32-element array $(0, 1, \ldots, 31)$ is distributed over four memory modules. The possible stride permutation accesses in this case are $P_{32,1}$, $P_{32,2}$, $P_{32,4}$, $P_{32,8}$, and $P_{32,16}$. The low order interleaving in Fig. 2(a) allows only conflict-free access for linear access, $P_{32,1}$. All the others introduce conflicts. The row-rotation and linear transformation schemes in Fig. 2(b) and (c), respectively, provide conflict-free access for $P_{32,1}$, $P_{32,2}$, and $P_{32,4}$. By noting that the elements 0, 8, and 16 are stored into the same module, we find that accesses $P_{32,8}$ and $P_{32,16}$ introduce conflicts. Especially the perfect shuffle access $P_{32,16}$ is difficult: the accesses should be performed in the following order: $([0, 16, 1, 17], [2, 18, 3, 19], [4, 20, 5, 21], [6, 22, 7, 23], [8, 24, 9, 25], [10, 26, 11, 27], [12, 28, 13, 29], [14, 30, 15, 31])$. In the previously reported access schemes, stride access has been defined to access every kth element while in perfect shuffle access $P_{N,N/2}$ elements with distances of 1 and $N/2$ need to be accessed. This example illustrates the principal difference of the stride access and stride permutation access.

3 Conflict-Free Parallel Memory Access for Stride Permutation

The previous discussion shows that there is need to perform several memory accesses in parallel. In addition, it was found that when an N-element data array is accessed according to a stride permutation, there will be need to support all the strides from 1 to $N/2$. The solution proposed in this paper will be based on linear transformations discussed in the following.

3.1 Linear Address Transformations

Linear transformations are based on modulo-2 arithmetic, thus the transformation can be expressed with binary matrices as

$$r = I_{n-q}(a_{n-1}, a_{n-2}, \ldots, a_{n-q})^T; \tag{6}$$
$$m = Ta = (T_H|T_L)\, a \tag{7}$$

where $a = (a_{n-1}, a_{n-2}, \ldots, a_0)^T$ is the index address referencing the element to be accessed, T is a module transformation matrix, T_L is the rightmost $q \times q$ square matrix in T and T_H is the remaining $q \times n - q$ matrix in T. It should be noted that in this representation the least significant bit of a is in the bottom of the vector. As an example, the matrix T used in Fig. 2(c) is

$$T = \begin{pmatrix} 0\,1\,0\,1\,0 \\ 0\,0\,1\,0\,1 \end{pmatrix}. \tag{8}$$

In [17], the basic requirement for the data distribution has been derived as follows.

Theorem 2. *An interleaved memory system has a unique storage location for each addressed element iff the matrix A_L has full rank.*

In [8], it is even suggested that T should have full rank and, in particular, the main diagonal of T should consists of 1's. Missing 1's in the main diagonal may result in poor performance for linear access, $P_{N,1}$. In addition, off-diagonal 1's complicate the construction of address generators.

In [1], stride accesses have been investigated with the aid of transformation periodicity referring to the minimum period of the sequence of module numbers generated when consecutive addresses are used as the input sequence. This results in the following requirement.

Theorem 3. *In matched memory system, $S = 2^s$ stride access over $Q = 2^q$ memories is conflict-free iff the linear transformation matrix T is*

a) periodic SQ and

b) $(a + iS)T \bmod Q = (a + jS)T \bmod Q$ only if $i \bmod Q = j \bmod Q$.

The condition a) guarantees that the access is conflict-free regardless of the array length and initial address of the array. The condition b) defines that each memory bank is referenced only once in Q parallel accesses. It can be shown that under the previous constraints a conflict-free access scheme supporting several strides cannot be designed [8].

3.2 Access Scheme for Stride Permutation

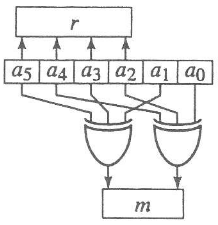
The previous discussion provides some hints to develop an access scheme for stride permutations. Although Theorem 3 suggests that a scheme cannot be designed, we may, however, relax the requirements by assuming that the array length is a constant and power-of-two, $N = 2^n$. This implies that constraints on the initial address need to be set; an array with length of 2^n should be stored in n-word boundaries. Such an constraint has already been used in several commercial DSP processors for performing circular addressing [18]. We also assume that the number of memory modules is a power-of-two $Q = 2^q$, which is actually a practical assumption in digital systems. In addition, we assume that also the strides in stride permutation access are powers-of-two, $S = 2^s$. This results in that the address mapping should produce a q-bit memory module address and a $(n - q)$-bit row address. Here the row address is formed as earlier in (6); row address $r = (r_{n-q-1}, r_{n-q-2}, \ldots, r_0)^T$ is obtained by extracting the $n - q$ most significant bits from the address:

$$r_i = a_{i+q}, i = 0, 1, \ldots, n - q - 1. \tag{9}$$

The previous assumptions define that the transformation matrices will be specific for each array length N and number of modules Q. However, the stride is not anymore a parameter for the matrix. Therefore, we introduce a new notation for the linear transformation matrix: $T_{N,Q}$, which defines clearly the array length and the number of modules.

The discussion in Section 2 related to the example in Fig. 2 implies that the periodicity of the linear transformation scheme used in the example is not large enough. This can be clearly seen by comparing the order of elements in each row; the ordering repeats after the fourth column, i.e., the period is 16. This is already reflected by the fact that the module address is generated by using four bits from the address.

The periodicity can be increased by adding the number of bits affecting the module address. This is already suggested in [9] for unmatched memory systems but the additional bit fields are only copied, not included into the bit-wise XOR operations. A special case of perfect shuffle access is discussed in [19], where two elements are accessed from a two-memory system, $Q = 2$. In such a case, the module address is defined by the parity of the address, thus the transformation



m	**0**	**1**	**2**	**3**
	0	1	2	3
	5	4	7	6
	10	11	8	9
	15	14	13	12
	17	16	19	18
	20	21	22	23
	27	26	25	24
	30	31	28	29
	34	35	32	33
	39	38	37	36
	40	41	42	43
	45	44	47	46
	51	50	49	48
	54	55	52	53
	57	56	59	58
	60	61	62	63

Fig. 5. Access scheme for 64-element array on 4-memory system corresponding to transformation matrix in (10): (a) module address generation and (b) contents of memory modules.

matrix $T_{2^n,2}$ is a vector of n elements of 1's. This implies that the additional bits should be included into the bit-wise XOR operations, i.e., each row in $T_{N,Q}$ should contain multiple 1's.

The use of diagonals have been suggested in [8] thus the obvious solution would be to add diagonals to T. Let us illustrate this approach with an example where 64-element array is distributed over four memory banks. In such a case, the transformation matrix $T_{64,4}$ would be the following:

$$T_{64,4} = \begin{pmatrix} 1\,0\,1\,0\,1\,0 \\ 0\,1\,0\,1\,0\,1 \end{pmatrix}. \tag{10}$$

This will result in the storage depicted in Fig. 5 and it is easy to see that all the stride permutation accesses with power-of-two strides from 1 to 32 are conflict-free. Performed computer simulations verified that the transformation matrix can be designed by filling the matrix with $q \times q$ diagonals in cases where n rem $q = 0$; transformation matrices $T_{k2^q,2^q}$ can be obtained by concatenating k identity matrices I_q.

The next question is how the matrix is formed when n rem $q \neq 0$. For this purpose, additional 1's need to be included into $T_{N,Q}$. In [8], such 1's where added as diagonals or antidiagonals off from the main diagonals. In [17], the main diagonals may contain 0's thus the additional 1's are spread over the matrix for fulfilling the full rank requirement. This results in the fact that rows may contain large number of ones, thus the number of bits needed in XOR-operations is increased. The effect is even worse in approach used in [7] where the rows may contain different number of 1's; one row is full of 1's, another contains only a single 1. This is extremely uncomfortable from the implementation point of view when several array lengths need to be supported since the transformation matrix will be different for different array length. In such a case, number of bits XOR'ed together varies from 1 to n.

The previous discussion implies that the additional bits should be concentrated to the right part of $T_{N,Q}$, i.e., to A_L in the original matrix T in (7). Such an arrangement eases the configuration of the address generation when the array length changes. If the 1's are in matrix A_H, the address bits a_i, which need to be included into XOR operations may change requiring multiplexing. Now, if all the configurations are performed for the least significant bits of a, these are always in the same position independent on the array length.

Therefore, in cases where n rem $q \neq 0$, we fill the transformation matrix with diagonals starting from the right lower corner and, if there is not enough space available in the left of the matrix, a partial diagonal is placed. The remaining partial diagonal wraps back to the right and filling is started from the rightmost column of $T_{N,Q}$ in a row, which is above the row where 1 in the leftmost column was placed. If the diagonal will hit the top row, it will be continued from the bottom row in the preceding column. All in all $(n + q - \gcd(q, n \bmod q))$ ones will be used in $T_{2^n,2^q}$ where $\gcd(\cdot)$ is the greatest common divisor. The entire access scheme can be formalized as follows

$$m_i = \bigoplus_{k=0}^{l_{n,q}(i)} a_{(jq+i) \bmod n}, \; i = 0, 1, \ldots, q-1 \;,$$
$$l_{n,q}(i) = \lfloor (n + q - \gcd(q, n \bmod q) - i - 1)/q \rfloor \tag{11}$$

where $\oplus$ denotes bit-wise XOR operation. The row address is obtained according to (9).

This approach provides a solution to the example shown in Fig. 2 and $T_{32,4}$ is the following

$$T_{32,4} = \begin{pmatrix} 0\,1\,0\,1\,1 \\ 1\,0\,1\,0\,1 \end{pmatrix} . \tag{12}$$

The contents of the memory modules stored according to $T_{32,4}$ is illustrated in Fig. 6. Once again it can be seen that the stride permutation access is supported for all the strides of powers-of-two from 1 to 16.

The proposed access scheme has been verified with computer simulations by generating storage organizations and verifying that each access is conflict-free. For a given array length $N = 2^n$, the number of memory modules Q was varied to cover all the possible numbers of powers-of-two, i.e., $Q = 2^0, 2^1, \ldots, 2^{n-1}$. For each parameter pair (N, Q), all the stride permutation accesses were performed with strides covering all the possible powers-of-two: $S = 2^0, 2^1, \ldots, 2^{n-1}$ and each parallel access was verified to be conflict-free. The power-of-two array lengths of were iterated from 2^1 to 2^{20}. The extensive simulation did not find any conflicts and, therefore, we can state that the proposed access scheme provides conflict-free parallel stride permutation access in practical cases, i.e., array lengths up to 2^{20}, for all the possible power-of-two strides on matched interleaved memory systems where the number of memory modules is a power-of-two.

m 0	1	2	3
0	3	2	1
7	4	5	6
10	9	8	11
13	14	15	12
19	16	17	18
20	23	22	21
25	26	27	24
30	29	28	31

Fig. 6. Contents of memory modules when a 64-element array is stored into a 4-memory system corresponding to transformation matrix in (12).

3.3 Address Generation

Before going into implementations, we may investigate the structure $T_{N,Q}$ when N is varied. In practical systems, Q is constant; the number of memory modules is only a design time parameter. As an example transformation matrices for 64-module systems are illustrated in Fig. 7 and few observations can be made from the structure of the matrices.

First, the matrices contain two principal diagonal structures: concatenated diagonals from the bottom-right corner to left and additional off-diagonals. The concatenated diagonals imply that the address a should be divided into q-bit fields and bit-wise XOR is performed between these fields. Since the concatenated diagonals in matrix for a shorter array is included in matrix for longer arrays, several array lengths can be supported easily; shorter arrays can be supported by feeding 0's to the most significant address bits.

The second observation is that the off-diagonals affect at most the $q-1$ least significant bits of the address a. In fact, (11) dictates that the number of 1's in off-diagonals is $q - \gcd(q, n \bmod q)$. In addition, the structure of off-diagonals depends on the relation between n and q but, since, in practice, q is constant, the structure depends on array length. However, there are only q different structures; the off-diagonal structure has periodic behavior when the array length is increasing. In Fig. 7, one complete period is shown and $T_{8192,64}$ would have the same off-diagonal structure as $T_{128,64}$.

The structure of off-diagonals implies that several array lengths can be supported if a predetermined control word configures additional hardware to perform the functionality of the off-diagonals. Such a configuration is actually simple by noting that the form of off-diagonals in different array lengths indicates rotation of least significant bits in a. The number of bits rotated is dependent on the relation between n and q.

According to the previous observations, the computation of the module address m can be interpreted as follows. First, the address a is divided into q-bit fields F^i starting from the least significant bit of a, i.e., $F^i = (a_{iq+q-1}, a_{iq+q-2}, \ldots, a_{iq+1}, a_{iq})^T$. If $e = n \bmod q > 0$, the e most significant bits of a exceeding

$$T_{128,64} = \begin{pmatrix} 0&1&1&0&0&0&0 \\ 0&0&1&1&0&0&0 \\ 0&0&0&1&1&0&0 \\ 0&0&0&0&1&1&0 \\ 0&0&0&0&0&1&1 \\ 1&0&0&0&0&0&1 \end{pmatrix}$$

$$T_{256,64} = \begin{pmatrix} 0&0&1&0&1&0&0&0 \\ 0&0&0&1&0&1&0&0 \\ 0&0&0&0&1&0&1&0 \\ 0&0&0&0&0&1&0&1 \\ 1&0&0&0&0&0&1&0 \\ 0&1&0&0&0&0&0&1 \end{pmatrix}$$

$$T_{512,64} = \begin{pmatrix} 0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&1 \end{pmatrix}$$

$$T_{1024,64} = \begin{pmatrix} 0&0&0&0&1&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0&0&1 \\ 1&0&0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&1&0&1&0 \\ 0&0&0&1&0&0&0&1&0&1 \end{pmatrix}$$

$$T_{2048,64} = \begin{pmatrix} 0&0&0&0&0&1&0&0&0&0&1 \\ 1&0&0&0&0&0&1&0&0&0&0 \\ 0&1&0&0&0&0&1&1&0&0&0 \\ 0&0&1&0&0&0&0&1&1&0&0 \\ 0&0&0&1&0&0&0&0&1&1&0 \\ 0&0&0&0&1&0&0&0&0&1&1 \end{pmatrix}$$

$$T_{4096,64} = \begin{pmatrix} 1&0&0&0&0&0&1&0&0&0&0&0 \\ 0&1&0&0&0&0&0&1&0&0&0&0 \\ 0&0&1&0&0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0&0&0&0&1 \end{pmatrix}$$

Fig. 7. Transformation matrices for module address generation for 64-memory systems.

the q-bit block border are extracted as a bit vector $L = (l_{e-1}, l_{e-2}, \ldots, l_0)^T$, which is

$$L = (a_{n-1}, a_{n-2}, \ldots, a_{n-e})^T . \tag{13}$$

Next, a q-bit field $X = (x_{q-1}, x_{q-2}, \cdots, x_0)$ is formed by extracting the $(q - \gcd(q, e))$ least significant bits of the address a and placing zeros to the most

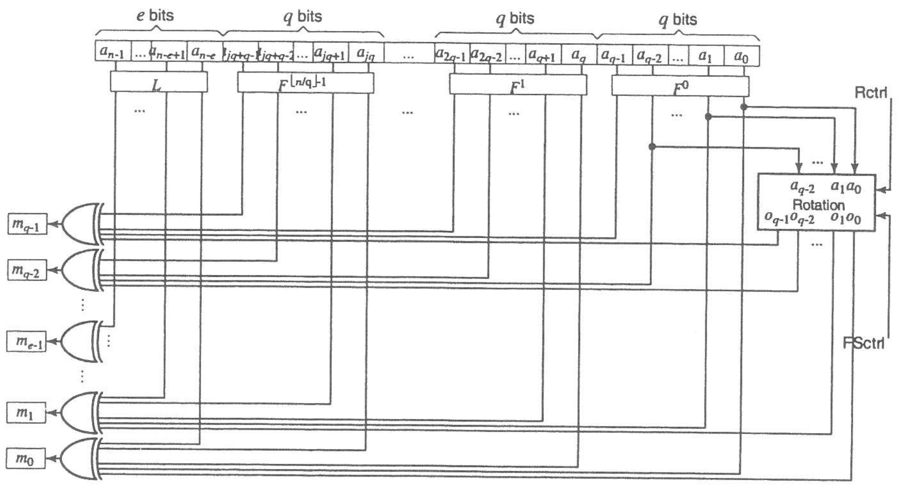

e bits
q bits
q bits
q bits
a_{n-1} ... a_{n-e+1} a_{n-e}
a_{jq+q-1} a_{jq+q-2} ... a_{jq+1} a_{jq}
a_{2q-1} a_{2q-2} ... a_{q+1} a_q
a_{q-1} a_{q-2} ... a_1 a_0
L
$F^{\lfloor n/q \rfloor -1}$
F^1
F^0
Rctrl
a_{q-2} $a_1 a_0$
Rotation
$o_{q-1} o_{q-2}$ $o_1 o_0$
FSctrl
m_{q-1}
m_{q-2}
m_{e-1}
m_1
m_0

Fig. 8. Principal block diagram of module address generation. Rctrl: rotation control. FSctrl: field selection control.

significant bits;

$$X = \left(0, \ldots, 0, a_{q-\gcd(q,e)-1}, \ldots, a_1, a_0\right)^T . \tag{14}$$

The bit vector X is rotated $g = (n - q \bmod q)$ bits to the left to obtain a bit vector $O = (o_{q-1}, \ldots, o_0)^T$, i.e.,

$$O = \mathrm{rot}_{(n-q) \bmod q}\,(X) \tag{15}$$

where $\mathrm{rot}_g(\cdot)$ denotes g-bit left rotation (circular shift) of the given bit vector, i.e.,

$$\begin{aligned} &\mathrm{rot}_g\ \left((a_{k-1}, a_{k-2}, \ldots, a_0)^T\right) = \\ &\quad \left(a_{k-g-1}, a_{k-g-2}, \ldots, a_0, a_{k-1}, \ldots, a_{k-g+1}, a_{k-g}\right)^T . \end{aligned} \tag{16}$$

Finally the module address m is obtained by performing bit-wise XOR operation between the vectors F_i, X, and L:

$$m_i = \begin{cases} o_i \oplus \left(\bigoplus_{j=0}^{\lfloor n/q \rfloor -1} a_{jq+i}\right), & i \geq e \\ l_i \oplus o_i \oplus \left(\bigoplus_{j=0}^{\lfloor n/q \rfloor -1} a_{jq+i}\right), & i < e \end{cases} . \tag{17}$$

A principal block diagram of the module address generation according to the previous interpretation is illustrated in Fig. 8. This block diagram contains a rotation unit shown in Fig. 9, which computes the vector O. This unit obtains $q-1$ least significant bits of a as an input and the $\gcd(q, e)-1$ most significant bits

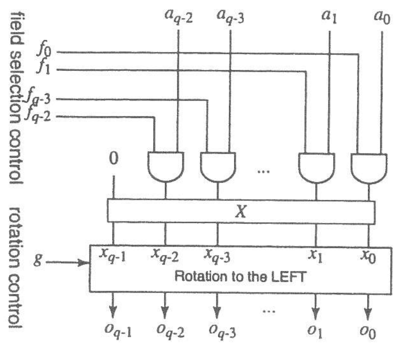


Fig. 9. Principal block diagram of rotation unit in module address generation.

of input are zeroed, thus the $q-\gcd(q,e)$ least significant bits are passed through, to form a q-bit vector $X = (0,\ldots,0,a_{q-\gcd(q,e)-1},a_{q-\gcd(q,e)-2},\ldots,a_1,a_0)^T$. These bits can be selected with the aid of a bit vector $f = (f_{q-2},\ldots,f_1,f_0))^T$, where the $q-\gcd(q,e)$ least significant bits are 1's and the $\gcd(q,e)-1$ most significant bits are 0's. When performing a bit-wise AND operation with the input vector and the obtained vector X is then rotated g bits to the left, the q-bit vector O is obtained.

The main advantage of the proposed scheme can be seen from the block diagram in Fig. 8. In the address generation, each individual XOR is performed on at most $\lfloor n/q \rfloor + 2$ bit lines while in other schemes, e.g., in [7], some XORs require all the n address bits, which complicates implementations when several array lengths need to be supported. The support for different array lengths in the proposed implementation requires only a single predetermined control word defining bit selection and rotation. However, this control word needs to be modified only when the length of the array to be accessed is changed.

4 Conclusions

In this paper, a conflict-free stride permutation access scheme for matched memory systems was proposed. It was assumed that 2^n data elements are distributed over 2^q independent memory modules. In this case, all the possible power-of-two stride permutation accesses are conflict-free, which was verified with computer simulations. The module address generation is simple requiring only bit-wise XOR operations.

The used assumptions dictate that the array length is constant and the initial address is zero. It was shown that several array lengths can be supported by including a q-bit left shifter into the module address generator. In this case all the additional operations are performed on the $q-1$ least significant bits of the address independent on the array length. The proposed scheme can support different initial addresses but arrays need to stored into n-word boundaries. However, this is not a strict requirement and such a restriction is already present in some addressing modes in commercial DSP processors.

Stride permutations are found in several DSP applications where small kernels are iterated thus a special addressing scheme supporting the access pattern provides advantage especially when long arrays are used, e.g., in FFT. The access scheme can also be used in application-specific array processors where operands need to be reordered according to stride permutation. In such cases, multi-ported memories can be avoided when an interleaved memory system is used. Furthermore, the proposed scheme can provide memory-efficiency since double buffering is avoided.

References

1. Harper III, D.T.: Increased memory performance during vector accesses through the use of linear address transformations. IEEE Trans. Comput. **41** (1992) 227–230
2. Budnik, P., Kuck, D.: The organization and use of parallel memories. IEEE Trans. Comput. **20** (1971) 1566–1569
3. Wijshoff, H.A.G., van Leeuwen, J.: The structure of periodic storage schemes for parallel memories. IEEE Trans. Comput. **34** (1985) 501–505
4. Deb, A.: Multiskewing - a novel technique for optimal parallel memory access. IEEE Trans. Parallel and Distrib. Syst. **7** (1996) 595–604
5. Frailong, J.M., Jalby, W., Leflant, J.: XOR-schemes: A flexible data organization in parallel memories. In: Proc. Int. Conf. Parallel Processing, St. Charles, IL, U.S.A. (1985)
6. Gao, Q.S.: The chinese remainder theorem and the prime memory system. In: Proc. Int. Conf. Computer Architecture, San Diego, CA, U.S.A. (1993) 337–340
7. Norton, A., Melton, E.: A class of boolean linear transformations for conflict-free power-of-two stride access. In: Proc. Int. Conf. Parallel Processing, St. Charles, IL, U.S.A. (1987) 247–254
8. Harper III, D.T.: Block, multistride vector, and FFT accesses in parallel memory systems. IEEE Trans. Parallel and Distrib. Syst. **2** (1991) 43–51
9. Valero, M., Lang, T., Peiron, M., Ayguadé, E.: Conflict-free access for streams in multimodule memories. IEEE Trans. Comput. **44** (1995) 634–646
10. Chen, S., Postula, A.: Synthesis of custom interleaved memory systems. IEEE Trans. VLSI Syst. **8** (2000) 74–83
11. Lin, H., Wolf, W.: Co-design of interleaved memory systems. In: Proc. Int. Workshop Hardware/Software Codesign, San Diego, CA, U.S.A. (2000) 46–50
12. Wuytack, S., Catthoor, F., de Jong, G., Man, H.J.D.: Minimizing the required memory bandwidth in VLSI system realizations. IEEE Trans. VLSI Syst. **7** (1999) 433–441
13. Granata, J., Conner, M., Tolimieri, R.: Recursive fast algorithms and the role of the tensor product. IEEE Trans. Signal Processing **40** (1992) 2921–2930
14. Davio, M.: Kronecker products and shuffle algebra. IEEE Trans. Comput. **30** (1981) 116–125
15. Astola, J., Akopian, D.: Architecture-oriented regular algorithms for discrete sine and cosine transforms. IEEE Trans. Signal Processing **47** (1999) 1109–1124
16. Takala, J., Akopian, D., Astola, J., Saarinen, J.: Constant geometry algorithm for discrete cosine transform. IEEE Trans. Signal Processing **48** (2000) 1840–1843

17. Sohi, G.S.: Interleaved memories for vector processors. IEEE Trans. Comput. **42** (1993) 34–44
18. Lapsley, P.D., Bier, J., Shoham, A., Lee, E.A.: DSP Processor Fundamentals: Architectures and Features. Berkeley Design Technology, Inc., Fremont, CA, U.S.A. (1996)
19. Cohen, D.: Simplified control of FFT hardware. IEEE Trans. Acoust., Speech, Signal Processing **24** (1976) 255–579

Highly Efficient Scalable Parallel-Pipelined Architectures for Discrete Wavelet Transforms

David Guevorkian, Petri Liuha, Aki Launiainen, and Ville Lappalainen

Nokia Research Center, Visiokatu-1, SF-33720, Tampere, Finland
`David.Guevorkian@nokia.com`

Abstract. Scalable parallel-pipelined architectures for discrete wavelet transforms (DWTs) are proposed based on their flowgraph representation. Architectures may be implemented with varying level of parallelism thus allowing of trading off between the cost (chip area and/or power consumption) and the performance (throughput or delay). At every level of parallelism (given hardware amount) the proposed architectures perform with approximately 100% of hardware utilization and demonstrate excellent area-time characteristics compared to the existing DWT designs. Architectures are regular and easy controlled, they do not contain feedback, long (depending on the length of the input) connection or switches and can be implemented as semisystolic arrays.

1 Introduction

The Discrete Wavelet Transform (DWT) [1]-[4] is an efficient technique for signal/image decomposition that has been studied and successfully applied to a wide range of applications: numerical analysis [5]-[6], biomedicine [7], different branches of image and video processing [1], [8]-[9], signal processing techniques [10], speech compression/decompression [11], etc. DWT based compression methods have become the basis of international standards such as JPEG 2000.

Since many applications need real-time computation of DWT, a number of ASIC architectures have already been proposed [12]-[25] for hardware implementation of DWTs. Among them are the low hardware complexity devices which require at least $2N$ clock cycles (cc's) to compute a DWT of a sequence having N samples (e.g., the devices proposed in [12]-[14], the architecture A2 in [15], *etc.*). Also a large number of devices, having a period of approximately N cc's, have been designed (e.g., the three architectures in [14] when they are provided with a doubled hardware, the architecture A1 in [15], the architectures in [16]-[18], the parallel filter in [19], etc.). Most of these architectures exploit the Recursive Pyramid Algorithm (RPA) [26] based on the tree-structured filter bank representation of DWTs (see Fig. 1). In this representation the input signal is processed by several (J) levels of decomposition (octaves) where the length of the processed signal is twice reduced from a level to level. Even though, pipelining has been employed to implement these structures, however, classical pipelined architectures use the same number of processing elements for every pipeline stage [12], [23]-[25]). Balancing of the pipeline stages is achieved by making the clocking

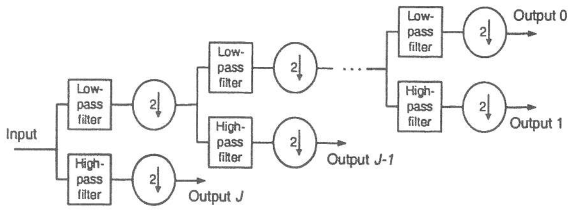


Fig. 1. Tree-structured flowgraph representation of a 1-D DWT

frequency twice smaller from a stage to stage (see, e.g., [23]-[25]). As a consequence of under-utilization of processing elements (PEs), the typical efficiency (i.e. hardware utilization) for these architectures strongly decreases with the number of decomposition levels. Approximately 100% of efficiency is achieved only in conventional architectures based on lifting scheme (see [4]) which, however, are either non-pipelined or employ only a restricted (two stage) pipelining [17], [18] and, in addition suffer from extensive either memory (chip area) or control requirements. The highest throughput achieved in known architectures is N clock cycles per N-point DWT. Twice faster performance is achieved in highly (about 100%) efficient architectures developed in [27]-[28] by including approximately twice lower number of PEs from a stage to stage.

In [29]-[30], *flowgraph representation* of DWTs (see examples on Figs. 2 and 3) has been suggested as a useful tool in designing parallel/pipelined DWT architectures. In particular, this representation fully reveals parallelism inherent to every octave as well as it clearly demonstrates data transfers within and between octaves. This allows to combine pipelining and parallelism to achieve a higher cost-efficient performance. This means implementing octaves in a pipelined mode where pipeline stages are parallelized at varying from stage to stage level. Incorporating varying level parallelism within pipeline stages allows to design parallel-pipelined devices with perfectly balanced pipeline stages.

In this work, general architectures of several DWT architectures operating at approximately 100% hardware utilization are proposed based on the flowgraph representation of DWTs shortly described in Section 2. The proposed structures may be implemented in different ways. In particular, they are scalable meaning that they can be implemented with varying level of parallelism giving opportunity to trade-off between the hardware complexity and performance. Several possible realizations of the proposed general structures are discussed in Section 3. The resulting architectures demonstrate excellent time and moderate area performance as compared to the conventional DWT architectures as follows from the discussion in Section 4. Throughputs of the architectures may vary between $NL/2^J$ time units (at the minimum level of parallelism) up to even one time unit (at the theoretical maximum level of parallelism) per an N-point DWT with J octaves and filters of the length L. The proposed architectures are regular and modular, easy controlled, and free of a feedback or a switch. They can be implemented as semisystolic arrays.

2 Flowgraph Representation and Parallel Algorithms for DWTs

There are several alternative definitions/representations of DWTs such as tree-structured filter bank, lattice structure, lifting scheme or matrix definitions [1]-[11]. In this section we use the matrix definition of the DWTs to arrive to their flowgraph representation which has been shown to be very efficient in designing efficient parallel/pipelined DWT architectures in [29]-[30]. The basic algorithm upon which the proposed structures are based is also described in this section.

Using the matrix definition, a discrete wavelet transform is a linear transform $y = H \cdot x$, where $x = [x_0, ..., x_{N-1}]^T$ and $y = [y_0, ..., y_{N-1}]^T$ are the input and the output vectors of length $N = 2^m$, respectively, and H is the DWT matrix of order $N \times N$ which is formed as the product of sparse matrices:

$$H = H^{(J)} \cdot ... \cdot H^{(1)}, \quad 1 \le J \le m; \quad H^{(j)} = \begin{pmatrix} D_j & 0 \\ 0 & I_{2^m - 2^{m-j+1}} \end{pmatrix}, \quad j = 1, ..., J \tag{1}$$

where I_k is the identity $(k \times k)$ matrix $(k = 2^m - 2^{m-j+1})$, and D_j is the analysis $(2^{m-j+1} \times 2^{m-j+1})$ matrix at stage j having the following structure:

$$D_j = \begin{pmatrix} l_1 & l_2 & ... & l_L & 0 & 0 & ... & 0 \\ 0 & 0 & l_1 & l_2 & ... & l_L & ... & 0 \\ & & & \ddots & & & & \\ l_3 & ... & l_L & 0 & 0 & ... & l_1 & l_2 \\ h_1 & h_2 & ... & h_L & 0 & 0 & ... & 0 \\ 0 & 0 & h_1 & h_2 & ... & h_L & ... & 0 \\ & & & \ddots & & & & \\ h_3 & ... & h_L & 0 & 0 & ... & h_1 & h_2 \end{pmatrix} = P_j \cdot \begin{pmatrix} l_1 & l_2 & ... & l_L & 0 & 0 & ... & 0 \\ h_1 & h_2 & ... & h_L & 0 & 0 & ... & 0 \\ 0 & 0 & l_1 & l_2 & ... & l_L & ... & 0 \\ 0 & 0 & h_1 & h_2 & ... & h_L & ... & 0 \\ & & & \ddots & & & & \\ l_3 & ... & l_L & 0 & 0 & ... & l_1 & l_2 \\ h_3 & ... & h_L & 0 & 0 & ... & h_1 & h_2 \end{pmatrix} \tag{2}$$

where $LP = [l_1, ..., l_L]$ and $HP = [h_1, ..., h_L]$ are the vectors of coefficients of the low-pass and high-pass filters, respectively, L is the length of the filters[1]), and P_j is the matrix of the perfect unshuffle operator of the size $(2^{m-j+1} \times 2^{m-j+1})$.

Adopting the representation (1)-(2), the DWT is computed in J stages (also called decomposition levels or octaves), where the jth stage, $j = 1, ..., J$, constitutes of multiplication of a sparse matrix $H^{(j)}$ by a current vector of scratch variables the first such vector being the input vector x. The corresponding algorithm can be written as the following pseudocode where $x_{LP}^{(j)} = \left[x_{LP}^{(j)}(0), ..., x_{LP}^{(j)}(2^{m-j} - 1)\right]^T$, and $x_{HP}^{(j)} = \left[x_{HP}^{(j)}(0), ..., x_{HP}^{(j)}(2^{m-j} - 1)\right]^T$, $j = 1, ..., J$, are $(2^{m-j} \times 1)$ vectors of scratch variables, and concatenation of column vectors $x_1, ..., x_k$ is denoted as $\left[(x_1)^T, ..., (x_k)^T\right]^T$.

[1] For clarity we assume both filters to have the same length which is an even number. The results are easily expanded to the general case of arbitrary filter lengths.

Algorithm 1.

1. Set $x_{LP}^{(0)} = \left[x_{LP}^{(0)}(0), ..., x_{LP}^{(0)}(2^m - 1)\right]^T = x$;

2a. For $j = 1, ..., J$ **compute**

$x_{LP}^{(j)} = \left[x_{LP}^{(j)}(0), ..., x_{LP}^{(j)}(2^{m-j} - 1)\right]^T$ and

$x_{HP}^{(j)} = \left[x_{HP}^{(j)}(0), ..., x_{HP}^{(j)}(2^{m-j} - 1)\right]^T$, where

$$\left[\left(x_{LP}^{(j)}\right)^T, \left(x_{HP}^{(j)}\right)^T\right]^T = D_j \cdot x_{LP}^{(j-1)}, \tag{3}$$

or, equivalently,

2b. For $i = 0, ..., 2^{m-j} - 1$,

Form the vector //* a subvector of length L of the vector $x_{LP}^{(j)}$ *//

$$\tilde{x} = \left[x_{LP}^{(j-1)}(2i), x_{LP}^{(j-1)}(2i+1), ..., x_{LP}^{(j-1)}((2i+L-1) \bmod 2^{m-j+1})\right]^T;$$

Compute

$$x_{LP}^{(j)}(i) = LP \cdot \tilde{x}; \quad x_{HP}^{(j)}(i) = HP \cdot \tilde{x};$$

3. Form the output vector $y = \left[x_{LP}^{(J)}, x_{HP}^{(J)}, x_{HP}^{(J-1)}, ..., x_{HP}^{(13)}, x_{HP}^{(12)}\right]^T$.

Computation of the Algorithm 1 with the matrices D_j of (2) can be clearly demonstrated using a flowgraph representation. An example for the case $N = 2^3 = 8, L = 4, J = 3$ is shown in Fig. 2. The flowgraph consists of J stages, the j-th stage, $j = 1, ..., J$, having 2^{m-j} nodes (depicted as boxes on Fig. 2). Each node represents a basic DWT operation (see Fig. 2(b)). The ith node, $i = 0, ..., 2^{m-j} - 1$, of the stage $j = 1, ..., J$ has incoming edges from L circularly consecutive nodes $2i, 2i+1, (2i+2) \bmod 2^{m-j+1} ..., (2i+L-1) \bmod 2^{m-j+1}$ of the preceding stage or (for the nodes of the first stage) from inputs. Every node has two outgoing edges. An upper (lower) outgoing edge represents the value of the inner product of the vector of low-pass (high-pass) filter coefficients with the vector of the values of incoming edges. Outgoing values of a stage are permuted according to the perfect unshuffle operator so that all the low-pass components are collected in the first half and the high-pass components are collected at the second half of the permuted vector. Low pass components are then forming the input to the following stage or (for the nodes of the last stage) represent output values. High-pass components represent output values at a given resolution.

Essentially, the flowgraph representation gives an alternative, rather demonstrative and easy-to-understand definition of discrete wavelet transforms. It has several advantages, at least from implementation point of view, as compared to the conventional DWT representations such as tree-structured filter bank, lifting scheme or lattice structure representation [30].

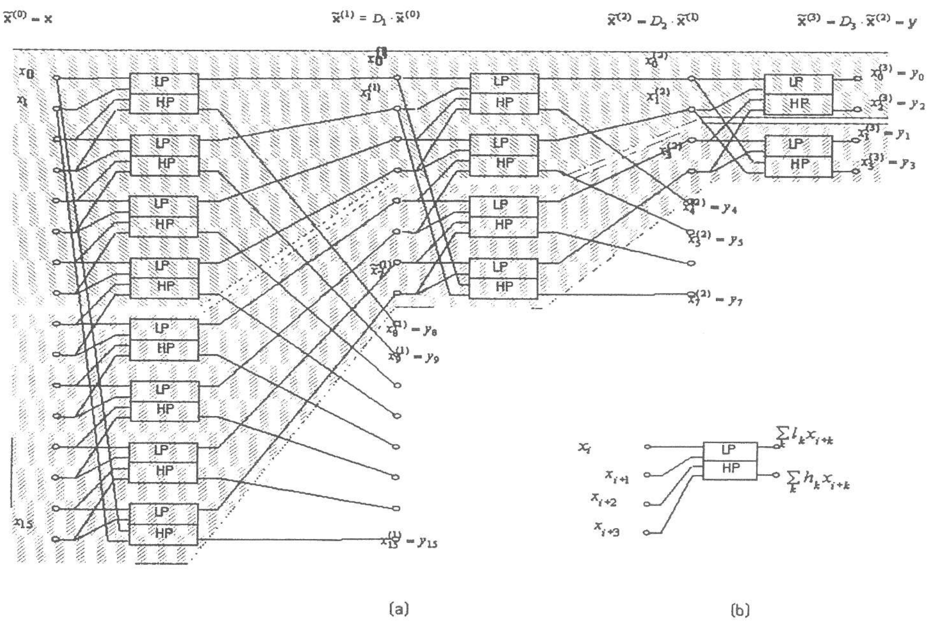


Fig. 2. Flowgraph representation of a 1-D DWT (*N=16, L=4, J=3*).

However, the flowgraph representation of DWTs as it has yet been presented has an inconvenience of being very large for bigger values of N. This inconvenience can be overcome based on the following observation. Assuming $J < \log_2 N$ (in the most of applications $J << \log_2 N$) one can see that the DWT flowgraph consists of $N/2^J$similar patterns (see the two hatching regions on Fig. 2). Every pattern can be considered as a 2^J-point DWT with a specific strategy of forming the input signals to its every octave. Merging the 2^{m-J} patterns in one, we can now obtain *compact (or core) flowgraph* representation of DWT. An example of a DWT compact flowgraph representation for the case $J = 3, L = 4$ is shown on Fig. 3. The compact DWT flowgraph has 2^{J-j} nodes at its j-th, stage, $j = 1, ..., J$, where now a set of 2^{m-J} temporarily distributed values are assigned to every node. Also, every outgoing edge corresponding to a high-pass filtering result of a node or low-pass filtering result of the node of the last stage represents a set of 2^{m-J} output values. Note that the structure of the compact DWT flowgraph does not depend on the length of the DWT but only on the number of decomposition levels and filter length. The DWT length is reflected only in the number of values represented by every node.

To illustrate computational process corresponding to the compact flowgraph let us adopt following notations. Let $\hat{D}_j$ be a matrix consisting of the first 2^{J-j+1} rows and the first $2^{J-j+1} + L - 2$ columns of D_j. For example, if $J - j + 1 = 2$ and $L = 6$, then $\hat{D}_j$ is of the form:

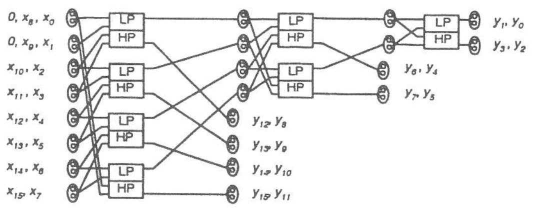


Fig. 3. Compact flowgraph representation of a 1-D DWT ($L = 4, J = 3$).

$$\hat{D}_j = \begin{pmatrix} l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & 0 & 0 \\ 0 & 0 & l_1 & l'_2 & l_3 & l_4 & l_5 & l_6 \\ h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & 0 & 0 \\ 0 & 0 & h_1 & h_2 & h_3 & h_4 & h_5 & h_6 \end{pmatrix}$$

Let us also conventionally divide the vector $x_{LP}^{(j-1)}$ of (3) into subvectors $x^{(j-1,s)} = x_{LP}^{(j-1)}(s \cdot 2^{J-j+1} : (s+1) \cdot 2^{J-j+1} - 1)$, $s = 0, ..., 2^{m-J} - 1$, where here and throughout the text the notation $x(a : b)$ stands for the subvector of x consisting of the a-th to b-th components of x. Then the input of the j-th, $j = 1, ..., J$, octave within the s-th compact DWT flowgraph is the subvector $\hat{x}^{(j-1,s)}\left(0 : 2^{J-j+1} + L - 3\right)$ of the vector

$$\hat{x}^{(j-1,s)} = \left[\left(x_{LP}^{(j-1,s \bmod 2^{m-J})}\right)^T, ..., \left(x_{LP}^{(j-1,(s+Q_j-1) \bmod 2^{m-J})}\right)^T\right]^T \tag{4}$$

being the concatenation of the vector $x_{LP}^{(j-1,s)}$ with the circularly next $Q_j - 1$ vectors where $Q_j = \lceil (L-2)/2^{J-j+1} \rceil$.

With these notations, the computational process represented by the compact flowgraph can be described with the following pseudocode.

```
Algorithm 2.
1. For $s = 0, ..., 2^{m-J} - 1$ set $x_{LP}^{(0,s)} = x(s \cdot 2^J : (s+1) \cdot 2^J - 1)$;
2. For $j = 1, ..., J$
   For $s = 0, ..., 2^{m-J} - 1$
   2.1. Set $\hat{x}^{(j-1,s)}$ according to (4)
   2.2. Compute $\left[\left(x_{LP}^{(j,s)}\right)^T, \left(x_{HP}^{(j,s)}\right)^T\right]^T = \hat{D}_j \cdot \hat{x}^{(j-1,s)}\left(0 : 2^{J-j+1} + L - 3\right)$,
3. Form the output vector $y = \left[\left(x_{LP}^{(J,0)}\right)^T, ..., \left(x_{LP}^{(J,2^{m-J}-1)}\right)^T, \left(x_{HP}^{(J,0)}\right)^T, ..., \left(x_{HP}^{(J,2^{m-J}-1)}\right)^T, ..., \left(x_{HP}^{(1,0)}\right)^T, ..., \left(x_{HP}^{(1,2^{m-J}-1)}\right)^T\right]^T$.
```

Implementing the cycle for s in parallel one can easily arrive to a parallel DWT realization. By exchanging the nesting order of cycles for j and s and implementing the (nested) cycle for j in parallel it is possible to arrive to a pipelined DWT realization. Both poorly parallel and poorly pipelined realizations would be inefficient since the number of operations is halved from an octave to the next one. However, combining the two methods we arrive to very efficient parallel-pipelined or partially parallel-pipelined realizations. The following pseudocode presents such parallel-pipelined DWT realization where we denote

$$s*(j)=\sum\nolimits_{n=1}^{j} Q_n. \tag{5}$$

```
Algorithm 3.
1. For $s=0,...,2^{m-J}-1$ set $x_{LP}^{(0,s)}=x(s\cdot 2^J:(s+1)\cdot 2^J-1)$;
2. For $s=s*(12),...,2^{m-J}+s*(J)-1$, For $j=J_1,...,J_2$ do in parallel
    2.1. Set $\hat{x}^{(j-1,s-s*(j))}$ according to (4)
    2.2. Compute
    $\left[\left(x_{LP}^{(j,s-s*(j))}\right)^T,\left(x_{HP}^{(j,s-s*(j))}\right)^T\right]^T=\hat{D}_j\cdot\hat{x}^{(j-1,s-s*(j))}\left(0:2^{J-j+1}+L-3\right).$
3. Form the output vector (See Step 3 of Algorithm 2.)
```

Note that computation of Algorithm 3 take place during the steps $s=s*(j),...,2^{m-J}+s*(j)-1$. At steps $s=s*(1),...,s*(2)-1$ computations of only the first octave are implemented, at steps $s=s*(2),...,s*(3)-1$ only operations of first two octaves are implemented, etc. In general, at the step $s=s*(1),...,2^{m-J}+s*(J)-1$ operations of the octaves $j=J_1,...,J_2$ are implemented where $J_1=\min\left\{j \text{ such that } s*(j)\geq s<s*(j)+2^{m-J}\right\}$ and $J_1=\max\left\{j \text{ such that } s*(j)\leq s<s*(j)+2^{m-J}\right\}$.

3 The Proposed DWT Architectures

In this section we present general structures of two types of DWT architectures, referred to as *Type 1 and Type 2 core DWT architectures,* as well as two other DWT architectures being constructed based on either core DWT architecture and referred to as multi-core DWT architecture and variable resolution DWT architecture, respectively. Both types of the core DWT architectures implement arbitrary discrete wavelet transform with J octaves with low-pass and high-pass filters having a length L not exceeding a given number $L_{\max}$.

The general structure representing both types of core DWT architectures is presented on Fig. 4 where dashed lines depict connections, which are present in Type 2 architectures but are absent in Type 1 architectures. In both cases the architecture consists of a data input block and J pipeline stages each stage containing a data routing block and a block of processor elements (PEs) wherein the data input block implements the Step 1 of the Algorithm 3, data routing

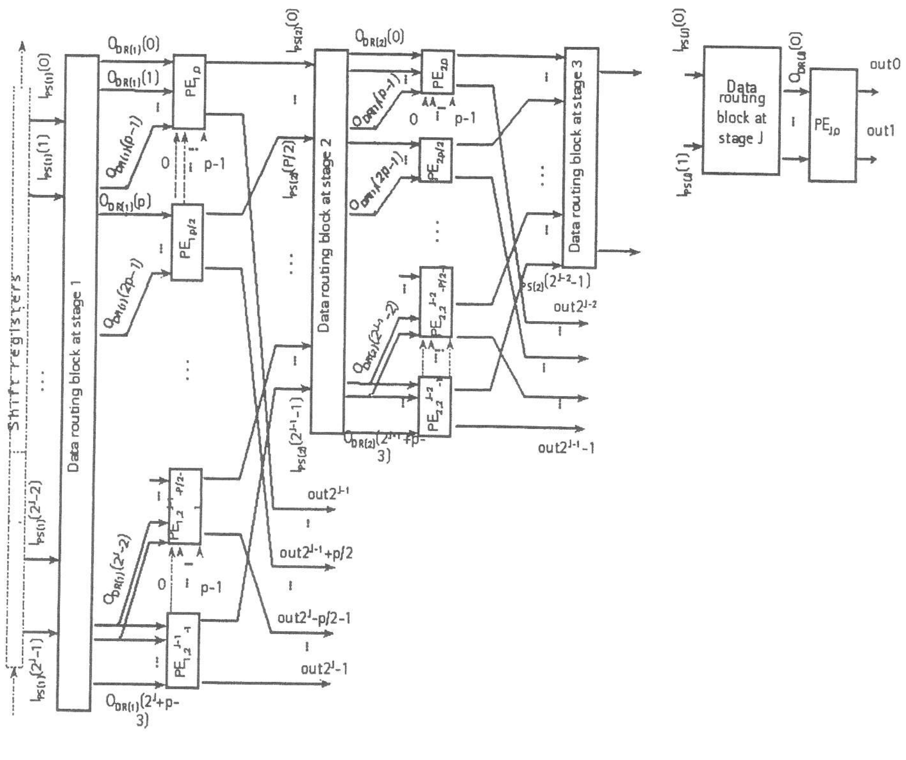


Fig. 4. The general structure of Type 1 and Type 2 core DWT architectures

blocks are responsible for Steps 2.1, and blocks of PEs are for computations of the Steps 2.2. The two types mainly differ by the possibility of data exchange between PEs of the same pipeline stage which are possible in Type 2 but not in Type 1 architectures.

The data input block of the core DWT architectures of both types may be realized as word-serial as well as word-parallel. In the former case the data input block consists of a single (word-serial) input port which is connected to a length-2^J shift register (dashed lined box on Fig. 4) having a word-parallel output from its every cell. In the latter case the data input block simply consists of 2^J parallel input ports. In both cases the data input block has 2^J parallel outputs connected to the 2^J inputs of the data routing block of the first pipeline stage.

3.1 Type 1 Core DWT Architecture

The basic operation of the Algorithm 3 (Step 2.2) is equivalent to 2^{J-j} pairs of vector-vector inner products:

$$x^{(j,s-s*(j))}(i) = LP \cdot \hat{x}^{(j-1,(s-s*(j))+)}(2i : 2i + L - 1),$$

$x^{(j,s-s*(j))}(i+2^{J-j}) = HP\cdot\hat{x}^{(j-1,(s-s*(j))+)}(2i:2i+L-1), i=0,...,2^{J-j}-1$

All the inner products may be implemented in parallel. On the other hand, every vector-vector inner product of length L can be obviously decomposed into a sequence of $L_p = \lceil L/p \rceil$ inner products of length p ($p \le L$) with accumulation of the results. With this, Algorithm 3 may be modified as follows.

```
Algorithm 3.1.
1. For $s = 0,...,2^{m-J}-1$ set $x_{LP}^{(0,s)} = x(s\cdot 2^J : (s+1)\cdot 2^J - 1)$;
2. For $s = s*(1),...,2^{m-J}+s*(J)-1$, For $j = J_1,...,J_2$ do in parallel
2.1. Set $\hat{x}^{(j-1,s-s*(j))}$ according to (4)
2.2. For $i = 0,...,2^{J-j}-1$ do in parallel
Set $S_{LP}(i) = 0$, $S_{HP}(i) = 0$;
For $n = 0,...,L_p-1$ do sequentially
```

$$S_{LP}(i) = S_{LP}(i) + \sum_{k=0}^{p-1} l_{np+k}\hat{x}^{(j-1,s-s*(j))}(2i+np+k); \qquad (6)$$

$$S_{HP}(i) = S_{HP}(i) + \sum_{k=0}^{p-1} h_{np+k}\hat{x}^{(j-1,s-s*(j))}(2i+np+k); \qquad (7)$$

```
Set $x_{LP}^{(j,s-s*(j))}(i) = S_{LP}(i)$; $x_{HP}^{(j,s-s*(j))}(i) = S_{HP}(i)$
3. Form the output vector (see Step 3 of Algorithm 2)
```

The general structure of the Type 1 core DWT architecture is presented by Fig. 4 where there are no connections depicted with dashed lines, that is there are no connections between PEs of a single stage. The architecture consists of a data input block (already described above) and J pipeline stages. In general, the jth pipeline stage, $j = 1,...,J$, of the Type 1 core DWT architecture consists of a data routing block having 2^{J-j+1} inputs $I_{PS(j)}(0),\dots,I_{PS(j)}(2^{J-j+1}-1)$ forming the input to the stage, and $2^{J-j+1}+p-2$ outputs $O_{DRB(j)}(0),\dots,O_{DRB(j)}(2^{J-j+1}+p-3)$ connected to the inputs of 2^{J-j} PEs. Every PE has p inputs and two outputs where $p \le L_{\max}$ is a parameter of the realisation describing the level of parallelism of every PE. Consecutive p outputs $O_{DRB(j)}(2i), O_{DRB(j)}(2i+1),...,O_{DRB(j)}(2i+p-1)$ of the data routing block of the jth, $j=1,...,J$, stage are connected to the p inputs of the ith, $i = 0,...,2^{J-j}-1$, PE ($PE_{j,i}$) of the same stage. First outputs of 2^{J-j} PEs of the jth pipeline stage, $j = 1,...,J-1$, form the outputs $O_{PS(j)}(0),\dots,O_{PS(j)}(2^{J-j}-1)$ of that stage and are connected to the 2^{J-j} inputs $I_{PS(j+1)}(0),\dots,I_{PS(j+1)}(2^{J-j}-1)$ of the data routing block of the next, $(j+1)$st, stage. The first output of the (one) PE of the last, Jth, stage is the 0th output $out(0)$ of the architecture. Second outputs of 2^{J-j} PEs of the jth pipeline stage, $j = 1,...,J$, form the (2^{J-j})th to $(2^{J-j+1}-1)$st outputs $out\,(2^{J-j}),...,out\,(2^{J-j+1}-1)$ of the architecture.

Let us now describe functionality of the blocks of the Type 1 core DWT architecture. For convenience, let us define a time unit as the period for PEs to complete their one operation (which is equal to the period between successive groups of p data to enter to the PE) and let us consider an operation step of the architecture to consist of L_p time units.

The data input block serially or in parallel accepts and in parallel outputs a group of components of the input vector at the rate of 2^J components per operation step. Thus, the vector $x_{LP}^{(0,s)}$ is formed on the outputs of the data input block at the step $s = 0, ..., 2^{m-J} - 1$.

The data routing block of the stage $j = 1, ..., J$, is a circuitry which at the first time unit $n = 0$ of its every operation step accepts in parallel a vector of 2^{J-j+1} components, and then at every time unit $n = 0, ..., L_p - 1$ of that operation step it outputs in parallel a vector of $2^{J-j+1} + p - 2$ components $np, np + 1, ..., (n+1)p + 2^{J-j+1} - 3$ of a vector being the concatenation (in the chronological order) of the vectors accepted at previous $\hat{Q}_j - 1$ steps, where

$$\hat{Q}_j = \lceil (L_{\max} - 2) / 2^{J-j+1} \rceil \quad j = 1, ..., J. \tag{8}$$

The functionality of the PEs used in the Type 1 core DWT architecture is to compute two inner products (6) and (7) of the vector on its p inputs with two vectors of predetermined coefficients during every time unit and to accumulate the results of both inner products computed during one operation step. At the end of every operation step, the two accumulated results pass to the two outputs of the PE and new accumulation starts. Possible structures of PEs for the Type 1 core DWT architectures are presented on Fig. 5 for the case of arbitrary p ,$p = 1$,$p = 2$, and $p = L_{\max}$, (Fig. 5, (a), (b), (c), and (d), respectively). These structures are for the "generic" DWT implementation independent of the filter coefficients. They can be easily optimized for specific filter coefficients.

It is easy to show that the architecture implements computations according to Algorithm 3.1 though with extra delay when $L < L_{\max}$. The extra delay is the consequence of the flexibility of the architecture for being able of implementing DWTs with arbitrary filter length $L \le L_{\max}$ while Algorithm 3.1 presents computation of a DWT with a fixed filter length L. In fact, the architecture is designed for the filter length $L_{\max}$ but also implements DWTs with shorter filters with a slightly increased time delay but without loosing in time period.

Denote

$$\hat{s}(0) = 0, \quad \hat{s}(j) = \sum\nolimits_{n=1}^{j} \hat{Q}_n, \quad j = 1, ..., J. \tag{9}$$

The delay between input and corresponding output vectors is equal to

$$T_d(C1) = \left(2^{m-J} + \hat{s}(J)\right) \lceil L/p \rceil \tag{10}$$

time units. The throughput or the time period (measured as the the intervals between time units when successive input vectors enter to the architecture) is equal to $T_p(C1)$ time units, where

$$T_p(C1) = 2^{m-J} \lceil L/p \rceil \tag{11}$$

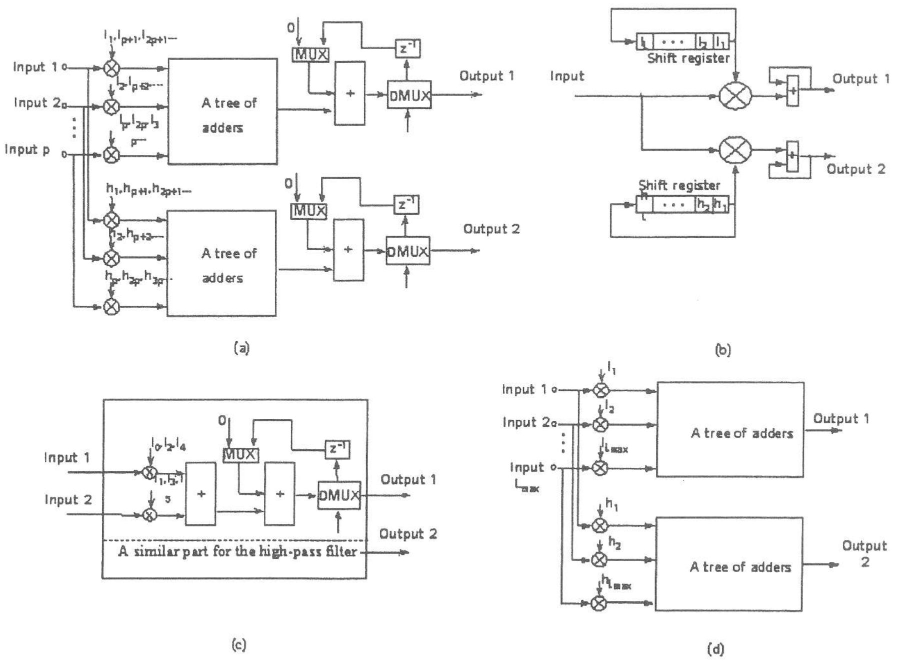


Fig. 5. Possible realizations of the PEs for the Type 1 core DWT architecture: (a) arbitrary p; (b) $p = 1$; (c) $p = 2$; (d) $p = L_{max}$.

From (10) and (11) we obtain that approximately 100% of efficiency (hardware utilization)[2] is achieved for the architecture both with respect to time delay or, moreover, time period complexities. A close efficiency is reached only in a few pipelined DWT designs (see [17], [27], [28]) whereas most of the known pipelined DWT architectures reach much less than 100% average efficiency. It should also be noted that a time period of at least $O(N)$ time units is required by known DWT architectures. The proposed architecture may be realized with varying level of parallelism depending on the parameter p. As follows from (11) the time period complexity of the implementation varies between $T_L(C1) = 2^{m-J}$ and $T_1(C1) = L2^{m-J}$. Thus the throughput of the architecture is $2^J/L$ to 2^J times faster than that of the fastest known architectures. The possibility of realization of the architecture with varying level of parallelism also gives an opportunity to trade-off the time and hardware complexities. It also should be noted that the architecture is very regular and needs an easy control (which is, essentially, a clock only) unlike, e.g. the architecture of [17]. It does not contain a feedback, a switch, or long connections depending on the size of the input but only

[2] The efficiency or hardware utilization is $E = (T(1) \cdot 100\%) \,/\, (K \cdot T(K))$ where $T(1)$ and $T(K)$ are the time complexities with one PE and with K PEs, respectively.

connections of the maximum of $O(L)$ length. Thus, it can be implemented as a semisystolic array.

3.2 Type 2 Core DWT Architecture

When implementing the basic operations (6) and (7) of Algorithm 3.1, multiplicands needed for the time unit $n = 1, ..., L_p - 1$ within the branch $i = 0, ..., 2^{J-j} - p/2 - 1$ can be obtained from the results obtained at step $n-1$ within the branch $i + p/2$. With this observation, the following modification of the basic algorithm may be derived. Denote

$$l'_k = \begin{cases} l_k \text{ for } k=0,...,p-1 \\ l_k/l_{k-p}, \text{ for } k=p,...,L-1 \end{cases}; \quad h'_k = \begin{cases} h_k \text{ for } k=0,...,p-1 \\ h_k/l_{k-p}, \text{ for } k=p,...,L-1 \end{cases}$$

Algorithm 3.2.

```
1. For $s = 0, ..., 2^{m-J} - 1$ set $x_{LP}^{(0,s)} = x(s \cdot 2^J : (s+1) \cdot 2^J - 1)$;
2. For $s = s*(1), ..., 2^{m-J} + s*(J) - 1$, For $j = J_1, ..., J_2$ do in parallel
2.1. Set $\hat{x}^{(j-1,s-s*(j))}$ according to (4)
2.2. For $i = 0, ..., 2^{J-j} - 1$ do in parallel
    For $k = 0, ..., p-1$
    { set $z(i,0,k) = l_k \hat{x}^{(j-1,s-s*(j))}(2i+k)$;
    Compute $S_{LP}(i) = \sum_{k=0}^{p-1} z_{LP}(i,0,k)$; $S_{HP}(i) = \sum_{k=0}^{p-1} z_{HP}(i,0,k)$; }
    For $n = 1, ..., L_p - 1$ do sequentially
        For $k = 0, ..., p-1$
        { set $z_{LP}(i,n,k) = \begin{cases} l'_{np+k} z(i+p/2, n-1, k) \text{ if } i<2^{J-j}-p/2 \\ l_{np+k} \hat{x}^{(j-1,s-s*(j))}(2i+k) \text{ if } i \geq 2^{J-j}-p/2 \end{cases}$;
        set $z_{HP}(i,n,k) = \begin{cases} h'_{np+k} z(i+p/2, n-1, k) \text{ if } i<2^{J-j}-p/2 \\ h_{np+k} \hat{x}^{(j-1,s-s*(j))}(2i+k) \text{ if } i \geq 2^{J-j}-p/2 \end{cases}$ }
        Compute $S_{LP}(i) = S_{LP}(i) + \sum_{k=0}^{p-1} z_{LP}(i,n,k)$; $S_{HP}(i) = S_{HP}(i) + \sum_{k=0}^{p-1} z_{HP}(i,n,k)$;
    Set $x_{LP}^{(j,s-s*(j))}(i) = S_{LP}(i)$; $x_{HP}^{(j,s-s*(j))}(i) = S_{HP}(i)$
3. Form the output vector (see Step 3 of Algorithm 2)
```

The general structure of the Type 2 core DWT architecture is presented on Fig. 4 where now the dashed lines showing connections between PEs of one stage are valid. Except for p inputs and two outputs (later on called main inputs and main outputs) every PE has now additional p inputs and p outputs (later on called intermediate inputs and outputs). The p intermediate outputs of $PE_{j,i+p/2}$ are connected to the p intermediate inputs of $PE_{j,i}$, $i = 0, ..., 2^{J-j} - p/2 - 1$. Other connections within the Type 2 core DWT architecture are similar to those within the Type 1 core DWT architecture.

Functionalities of the blocks of the Type 2 core DWT architecture are also similar to those of the Type 1 core DWT architecture. The difference is only in the functionality of PEs which at every time unit $n = 0, ..., L_p - 1$ of every

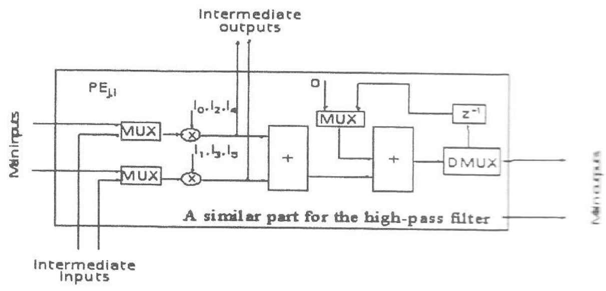


Fig. 6. A possible realization of a PE for the Type 2 core DWT architecture, $p = 2$

operation step is to compute two inner products of a vector, say, x, on its p either main or intermediate inputs with two vectors of predetermined coefficients, say LP' and HP' of length p as well as to compute a point-by-point product of x with LP'. At the time unit $n = 0$ the vector x is the one formed on the main p inputs of the PE and at time units $n = 1, ..., L_p - 1$ it is the one formed on the intermediate inputs of the PE. Results of both inner products computed during one operation step are accumulated and are passed to the two main outputs of the PE while the results of the point-by-point products are passed to the intermediate outputs of the PE. A possible structure of PEs for the Type 2 core DWT architecture for the case of $p = 2$ is presented on Fig. 6. Structures for arbitrary p and for $p = 1$, $p = 2$, and $p = L_{\max}$, can be easily designed similar to those on Fig.5.

Similar to as in the case of the Type 1 core DWT architecture one can see that the Type 2 core DWT architecture implements Algorithm 3.2 with time delay and time period characteristics given by (10) and (11). The other characteristics of these two architectures are also similar. In particular, it is very fast, it may be implemented as a semisystolic architecture and with varying level of parallelism giving opportunity of trade-off between time and hardware complexities. The difference between these two architectures is that the shift registers of data routing blocks of the Type 1 core DWT architecture are replaced with additional connections between PEs within the Type 2 core DWT architecture.

3.3 Multi-core DWT Architectures

The two types of core DWT architectures described above may be implemented with varying level of parallelism depending on the parameter p. Further flexibility in the level of parallelism is achieved within multi-core DWT architectures by introducing a new parameter $r = 1, ..., 2^{m-J}$. The multi-core DWT architecture is, in fact obtained from corresponding (single-)core DWT architecture by expanding it r times. Thus one can again consider Fig. 4 for the general structure of multicore architectures but the numbers of PEs at every pipeline stage should be multiplied by r.

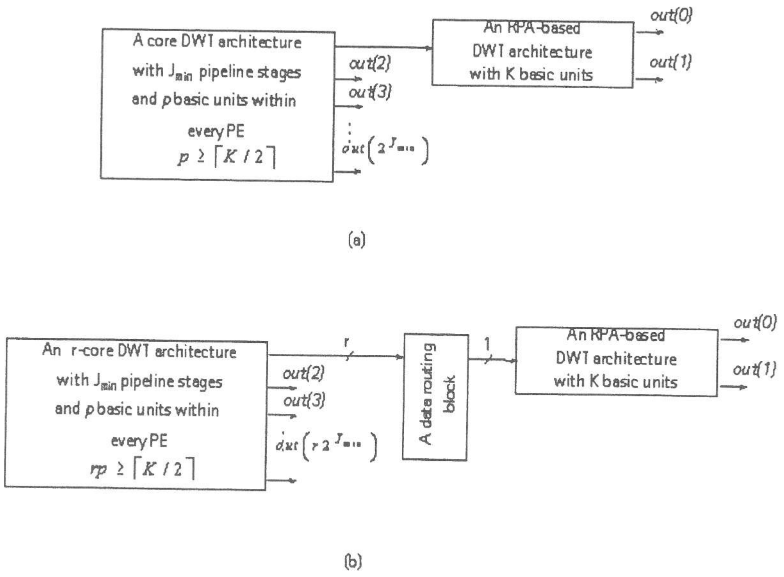

A core DWT architecture with J_{min} pipeline stages and p basic units within every PE $p \geq \lceil K/2 \rceil$
out(2)
out(3)
$out(2^{J_{min}})$
An RPA-based DWT architecture with K basic units
out(0)
out(1)
(a)
An r-core DWT architecture with J_{min} pipeline stages and p basic units within every PE $rp \geq \lceil K/2 \rceil$
r
out(2)
out(3)
$out(r2^{J_{min}})$
A data routing block
1
An RPA-based DWT architecture with K basic units
out(0)
out(1)
(b)

Fig. 7. The variable resolution DWT architecture: (a) based on a single core DWT architecture; (b) based on a multi-core DWT architecture.

The both types of multi-core DWT architectures are r times faster than the (single-)core DWT architectures, that is a linear speed-up with respect to the parameter r is achieved. The delay between input and corresponding output vectors is equal to

$$T_d(C1) = \left(2^{m-J} + \hat{s}(J)\right) \lceil L/p \rceil / r \tag{12}$$

time units and the throughput or the time period is equal to

$$T_p(C1) = 2^{m-J} \lceil L/p \rceil / r \tag{13}$$

time units. Thus further speed-up and flexibility for trade-off between time and hardware complexities is achieved within multi-core DWT architectures. Architectures are modular and regular and may be implemented as semisystolic arrays. As a possible realisation of the multi-core DWT architecture for the case of $p = L = L_{\max}$ and $r = 2^{m-J}$ one can consider the DWT flowgraph itself (see Fig. 2) where nodes (rectangles) should be considered as PEs and small circles as latches. This example of realization has been reported in [29]-[30] where it was referred to as fully-parallel pipelined (FPP) architecture.

3.4 Variable Resolution DWT Architectures

The above-described architectures implement DWTs with the number of octaves not exceeding a given number J. They may implement DWTs with smaller than

J number of octaves though with some loss in hardware utilisation. The variable resolution DWT architecture implements DWTs with arbitrary number J' of octaves whereas the efficiency of the architecture remains approximately 100% whenever J' is larger than or equal to a given number $J_{\min}$.

The general structure of the variable resolution DWT architecture is shown on Fig. 7,(a). It consists of a core DWT architecture corresponding to $J_{\min}$ decomposition levels and an arbitrary serial DWT architecture, for, instance, an RPA-based one ([14]-[17], [19]-[20], [22]). The core DWT architecture implements the first $J_{\min}$ octaves of the J'-octave DWT. The low-pass results from the *out(0)* of the core DWT architecture are passed to the serial DWT architecture. Then the serial DWT architecture implements the last $J' - J_{\min}$ octaves of the J'-octave DWT. Since the core DWT architecture may be implemented with varying level of parallelism it can be balanced with the serial DWT architecture in such a way that approximately 100% of hardware utilisation is achieved whenever $J' \geq J_{\min}$.

To achieve the balancing between the two parts the core DWT architecture must implement a $J_{\min}$-octave N-point DWT with the same throughput or faster as the serial architecture implements $(J' - J_{\min})$-octave M-point DWT $(M = (N/2^{J_{\min}}))$. Serial architectures found in the literature implement a M-point DWT either in $2M$ time units ([14], [15}) or in M time units ([14]-[19]) correspondingly employing either L or $2L$ basic units (BUs, multiplier-adder pairs). They can be scaled down to contain arbitrary number $K \leq 2L$ BUs so that an M-point DWT would be implemented in $M \lceil 2L/K \rceil$ time units. Since the (Type 1 or Type 2) core DWT architecture implements a $J_{\min}$-octave N-point DWT in $N \lceil L/p \rceil / 2^{J_{\min}}$ time units the balancing condition becomes $\lceil L/p \rceil \leq \lceil 2L/K \rceil$ which will be satisfied if $p = \lceil K/2 \rceil$. With this condition the variable resolution DWT architecture will consist of totally

$$A = 2p\left(2^{J_{\min}} - 1\right) + K = \begin{cases} K2^{J_{\min}}, \text{ if } K \text{ is even} \\ (K+1)2^{J_{\min}} - 1, \text{ if } K \text{ is odd} \end{cases}$$

BUs and will implement a J'-octave N-point DWT in T_d time units, where

$$T_d = N \lceil 2L/K \rceil / 2^{J_{\min}}$$

A variable resolution DWT architecture based on a multi-core DWT architecture may also be constructed (see Fig. 7,(b)) where now a data routing block is inserted between the multi-core and serial DWT architectures. The functionality of the data routing block is to parallelly accept and serially output digits at the rate of r samples per operation step. The balancing condition in this case is $rp = \lceil K/2 \rceil$, and the area time characteristics are

$$A = 2pr\left(2^{J_{\min}} - 1\right) + K = \begin{cases} K2^{J_{\min}}, \text{ if } K \text{ is even} \\ (K+1)2^{J_{\min}} - 1, \text{ if } K \text{ is odd} \end{cases}, \quad T_d = N \lceil 2L/K \rceil / 2^{J_{\min}}$$

3.5 Conclusions and a Summary of the Performance

Table 1 presents a comparative performance of the proposed architectures with some conventional architectures. In this table, as it is commonly accepted in the literature,

Table 1. Comparative performance of some DWT architectures

Architecture	Area, A (number of BUs)	Period, T_p	AT_p^2
Architectures in [14], [15]	L	$2N$	$4N^2L$
Architectures in [14]-[19]	$2L$	N	$2N^2L$
Architectures in [12],[24]	JL	N	JN^2L
Architectures of [27],[28]	$4L$or $\sum_{j=1}^{J} \lceil L/2^{j-2} \rceil$	$N/2$	$\approx N^2L$
FPP DWT [29]-[30] (pipelined)	$2NL(1-1/2^J)$	1 (per vector)	$2NL(1-1/2^J)$
LPP DWT [29]-[30]	$2L\left(2^J-1\right)$	$N/2^J$	$N^2L(2^J-1)/2^{2J-1}$ $\approx N^2L/2^{J-1}$
Single-core DWT (Type 1 or 2)	$2p\left(2^J-1\right)$	$N\lceil L/p \rceil/2^J$	$\approx N^2p\lceil L/p \rceil^2/2^{J-1}$
Single-core DWT, p=1	$2\left(2^J-1\right)$	$NL/2^J$	$\approx N^2L^2/2^J$
Single-core DWT p=L$_{max}$ $(L \leq L_{\max})$	$2L_{\max}\left(2^J-1\right)$	$N/2^J$	$\approx N^2L_{\max}/2^{J-1}$
Multi-core DWT	$2pr\left(2^J-1\right)$	$(N[L/p])/\left(r2^J\right)$	$\approx$ $\left(N^2p[L/p]^2\right)/\left(r2^{J-1}\right)$
Multi-core DWT, r=4, p=1	$8\left(2^J-1\right)$	$NL/2^{J+2}$	$\approx N^2L^2/2^{J+1}$
Multi-core DWT r=4, p=L$_{max}$$(L \leq L_{\max})$	$2rL_{\max}\left(2^J-1\right)$	$N/\left(r2^J\right)$	$\approx \left(N^2L_{\max}\right)/\left(r2^{J-1}\right)$
Variable resolution single-core DWT $p \geq \lceil K/2 \rceil$ $(K \leq 2L)$	$2p\left(2^{J_{\min}}-1\right)+K \approx K2^{J_{\min}}$	$N\lceil 2L/K \rceil/2^{J_{\min}}$ $\approx$ $2NL/(K2^{J_{\min}})$	$\approx \frac{N^2L^2}{K2^{J_{\min}-2}}$

the area of the architectures was counted as the number of used multiplier-adder pairs which are the basic units (BUs) in DWT architectures. The time unit is counted as time period of one multiplication since this is the critical pipeline stage. Characteristics of the DWT architectures proposed in this paper (the last seven rows in Table 1) are given as for arbitrary realisation parameters $L_{\max}$, p, and r as well as for some examples of parameter choices. It should be mentioned that the numbers of BUs used in the proposed architectures are given assuming the PE examples of Figs 5 and 6. (where PE with p inputs contains $2p$ BUs). However, PEs could be further optimized to involve less number of BUs.

As follows from Table 1, the proposed architectures, compared to the conventional ones, demonstrate excellent time characteristics at moderate area requirements. Advantages of the proposed architectures are best seen when considering the performances with respect to AT_p^2 criterion, which is commonly used to estimate performances of high-speed oriented architectures. Architectures presented in the first two rows of Table 1 are either non-pipelined or restricted (only two stage) pipelined ones and they operate at approximately 100% hardware utilisation as is the case for our proposed architectures. So their performance is "proportional" to the performance of our architectures which however are much more flexible in the level of parallelism resulting in a wide range of time and area complexities. The third row of Table 1 presents J stage pipelined architectures with a poor hardware utilization and consequently a poor performance. The fourth to sixth rows of Table 1 present architectures from our previous publications which are J stage pipelined and achieve 100% hardware utilization and good performance but do not allow a flexible range of area and time complexities as the architectures proposed in this paper do.

References

1. S. G. Mallat, "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 2, n. 12, Dec. 1989, pp. 674-693.
2. M. Vetterli and J. Kovacevic, Wavelets and Subband Coding, Englewood Cliffs (NJ): Prentice-Hall, 1995.
3. I. Daubachies, Ten Lectures on Wavelets, Philadelphia (PA): SIAM, 1992.
4. W. Sweldens, "The Lifting Scheme: A New Philosophy in Biorthogonal Wavelets Constructions," Proc. SPIE Conf., 1995, vol. 2569, pp.68-79.
5. G. Beylkin, R. Coifman, and V. Rokhlin, Wavelet in Numerical Analysis in Wavelets and their Applications, New York (NY): Jones and Bartlett, 1992, pp.181-210.
6. G. Beylkin, R. Coifman, and V. Rokhlin, Fast Wavelet Transforms and Numerical Algorithms, New Haven (CT): Yale Univ., 1989.
7. L. Senhadji, G. Carrault, and J.J. Bellanguer, "Interictal EEG Spike Detection: A New Framework Based on The Wavelet Transforms," in Proc. IEEE-SP Int. Symp. Time-Frequency Time-Scale Anal., Philadelphia (PA) Oct. 1994, pp.548-551.
8. S. G. Mallat, "Multifrequency Channel Decompositions of Images and Wavelet Models," IEEE Trans. on Acoust., Speech, Signal Processing, vol. 37, n. 12, 1989, pp. 2091-2110.
9. Z. Mou and P. Duhamel, "Short-Length FIR Filters and Their Use in Fast Non Recursive Filtering," IEEE Trans. on Signal Processing, vol. 39, n.6, Jun. 1991, pp. 1322-1332.
10. A. N. Akansu and R. A. Haddad, Multiresolution Signal Decomposition: Transforms, Subbands and Wavelets, New York (NY): Academic, 1992.
11. R. Kronland-Martinet, J. Morlet, and A. Grossman, "Analysis of Sound Patterns through Wavelet Transform," Int. Journ. Pattern Recognitiona and Artificial Intell., vol. 1, n.2, 1987, pp.273-302.
12. S. B. Pan, and R. H. Park, "New Systolic Arrays for Computation of the 1-D Discrete Wavelet Transform," Proc., IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, 1997, Vol. 5, 1997 , pp. 4113-4116.
13. A. B. Premkumar, and A. S. Madhukumar, "An Efficient VLSI Architecture for the Computation of 1-D Discrete Wavelet Transform," Proc., IEEE Int. Conf. On Information, Communications and Signal
14. M. Vishwanath, R. M. Owens, and M. J. Irwin, "VLSI Architectures for the Discrete Wavelet Transform," IEEE Trans. on Circuits and Systems-II: Analog and Digital Signal Processing, vol. 42, n. 5, May 1995, pp.305-316.

15. J. Fridman, and S. Manolakos, "Discrete Wavelet Transform: Data Dependence Analysis and Synthesis of Distributed Memory and Control Array Architectures," IEEE Trans. on Signal Processing, vol. 45, n. 5, May 1997, pp. 1291-1308.
16. K. K. Parhi, and T. Nishitani, "VLSI Architectures for Discrete Wavelet Transforms," IEEE Trans. on VLSI Systems, vol. 1, n.2, 1993, pp. 191-202.
17. T. C. Denk, and K. K. Parhi, "Systolic VLSI Architectures for 1-D Discrete Wavelet Transform," Proc. Thirty-Second Asilomar Conf. on Signals, Systems & Computers, 1998, vol. 2 pp. 1220-1224.
18. G. Knowles, "VLSI Architecture for the Discrete Wavelet Transform," Electronics Letters, vol. 26, n. 15, 1990, pp. 1184-1185.
19. C. Chakrabarti and M. Vishwanath, "Efficient Realizations of Discrete and Continuous Wavelet Transforms: From Single Chip Implementations to Mappings on SIMD Array Computers," IEEE Trans. on Signal Processing, vol. 43, n. 3, 1995, pp. 759-771.
20. C. Chakrabarti, M. Vishwanath, and R.. M. Owens, "Architectures for Wavelet Transforms: A Survey," Journal of VLSI Signal Processing, vol. 14, n. 2, 1996, pp. 171-192.
21. A. Grzeszczak, M. K. Mandal, S. Panchanatan, "VLSI Implementation of Discrete Wavelet Transform," IEEE Trans. on VLSI Systems, vol. 4, n. 4, Dec. 1996, pp. 421-433.
22. M. Vishwanath, and R.. M. Owens, "A Common Architecture for the DWT and IDWT," Proc., IEEE Int. Conf. on Application Specific Systems, Architectures and Processors, 1996, pp. 193-198.
23. C. Yu, C. H. Hsieh, amd S. J. Chen, "Desing and Implementation of a Highly Efficient VLSI Architecture for Discrete Wavelet Transforms," Proc., IEEE Int. Conf. on Custom Integrated Circuits, 1997, pp. 237-240.
24. S. B. Syed, M. A. Bayoumi, "A Scalable Architecture for Discrete Wavelet Transform," Proc., Computer Architectures for Machine Perception (CAMP '95), 1995, pp. 44 -50.
25. F. Marino, "A 'Double-Face' Bit-Serial Architecture for the 1-D Discrete Wavelet Transform", IEEE Trans. on Circuits and Systems II - Analog and Digital Signal Processing, vol. 47, n. 1, Jan. 2000, pp. 65-71..
26. M. Vishwanath, "The Recursive Pyramid Algorithm for the Discrete Wavelet Transform," IEEE Trans. on Signal Processing, vol. 42, n. 3, 1994, pp. 673-677.
27. F. Marino, D. Gevorkian, and J. Astola, "Highly efficient High-Speed/Low-Power Architectures for the 1-D DWT," IEEE Trans. on Circuits and Systems II, Analog and Digital Signal Processing, vol. 47, No 12, 2000, pp. 1492-1502.
28. F. Marino, D. Gevorkian, and J. Astola, "High-Speeed/Low-Power 1-D DWT Architectures with high efficiency," Proc., IEEE Int. Conf. on Circuits and Systems, May 28-31, Geneva, Switzerland, vol. 5, pp. 337-340.
29. D. Gevorkian, F. Marino, S. Agaian, and J. Astola, "Highly efficient fast architectures for discrete wavelet transforms based on their flowgraph representation," Proc., Int. Conf. EUSIPCO-2000, Sept. 2000, Tampere, Finland.
30. D. Gevorkian, F. Marino, S. Agaian, and J. Astola, "Flowgraph representation of discrete wavelet transforms and wavelet packets for their efficient parallel implementation," Proc., TICSP Int. Workshop on Spectral Transforms and Logic Design for Future Digital Systems, June 2-3, 2000, Tampere, Finland.

Automatic VHDL Model Generation of Parameterized FIR Filters

E. George Walters III[1], John Glossner[2], and Michael J. Schulte[1]

[1] Computer Architecture and Arithmetic Laboratory, Computer Science and Engineering Department, Lehigh University, Bethlehem PA 18015 USA
[2] Sandbridge Technologies, 1 N Lexington Ave, 10th Floor, White Plains NY 10601

Abstract. This paper describes a Java-based tool that automatically generates structural level VHDL models of FIR Filters. Automatic generation of VHDL models allows the designer to rapidly explore the design space and test the impact of parameters on the design. The tool is based on a general purpose computer arithmetic component package developed at Lehigh University and can easily be extended to enable rapid prototyping of other hardware accelerators used in embedded systems. In this paper, we describe the effects of truncated multipliers in FIR filters. We show that a 22.5 % reduction in area can be achieved for a 24-tap filter with 16-bit coefficients, and that the reduction error SNR is only 2.4 dB less than the roundoff error SNR of the same filter with no truncation. Using the techniques presented in this paper, the average reduction error of the filter is several orders of magnitude less than the average reduction error of the individual multipliers.

1 Introduction

The design of hardware accelerators for embedded systems presents many design tradeoffs that are difficult to quantify without bit-accurate simulation and area and delay estimates of competing alternatives. Structural level VHDL models can be used to evaluate and compare designs, but require significant effort to generate.

This paper presents a tool that was developed to evaluate the tradeoffs involved in using truncated multipliers in FIR filters. The tool is based on a package of Java classes that models the building blocks of computational systems, such as adders and multipliers. These classes generate VHDL descriptions, and are used by other classes in hierarchical fashion to generate VHDL descriptions of more complex systems. This paper describes the generation of truncated FIR filters as an example.

Previous techniques for modeling and designing digital signal processing systems with VHDL are presented in [1–5]. The tool described in this paper differs from those techniques by leveraging the benefits of object oriented programming (OOP). By subclassing existing objects, such as multipliers, the tool is easily extended to generate VHDL models that incorporate the latest optimizations and techniques.

Sections 1.1 and 1.2 provide background necessary for understanding the two's complement truncated multipliers used in the FIR filter architecture, which is described in Section 2. Section 3 describes the tool for automatically generating VHDL models of those filters. Synthesis results of specific filter implementations are presented in Section 4, with concluding remarks given in Section 5.

1.1 Two's Complement Multipliers

Parallel tree multipliers form a matrix of partial product bits, which are then added to produce a product. Consider an m-bit multiplicand, A, and an n-bit multiplier, B. If A and B are integers in two's complement form, then

$$A = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \qquad \text{and} \qquad B = -b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j \ . \tag{1}$$

Multiplying A and B together yields the following expression:

$$\begin{aligned} A \cdot B = a_{m-1}b_{n-1}2^{m+n-2} &+ \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} \\ - \sum_{i=0}^{m-2} b_{n-1}a_i 2^{i+n-1} &- \sum_{j=0}^{n-2} a_{m-1}b_j 2^{j+m-1} \ . \end{aligned} \tag{2}$$

The first two terms in (2) are positive. The third term is either zero (if $b_{n-1} = 0$) or negative with a magnitude of $\sum_{i=0}^{m-2} a_i 2^{i+n-1}$ (if $b_{n-1} = 1$). Similarly, the fourth term is either zero or a negative number. To produce the product of $A \times B$, the first two terms are added "as is". Since the third and fourth terms are negative (or zero), they are added by complementing each bit, adding '1' to the LSB column, and sign extending with a leading '1'. With these substitutions, the product is computed without any subtractions as:

$$\begin{aligned} P = a_{m-1}b_{n-1}2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} + \sum_{i=0}^{m-2} \overline{b_{n-1}a_i} 2^{i+n-1} \\ + \sum_{j=0}^{n-2} \overline{a_{m-1}b_j} 2^{j+m-1} + 2^{m+n-1} + 2^{n-1} + 2^{m-1} \ . \end{aligned} \tag{3}$$

Figure 1 shows the multiplication of two 8-bit integers in two's complement form. The partial product bit matrix is described by (3), and is implemented using an array of AND and NAND gates. The matrix is then reduced using techniques such as Wallace [6], Dadda [7], or Reduced Area reduction [8].

1.2 Truncated Multipliers

Truncated $m \times n$ multipliers, which produce results less than $m + n$ bits long, are described in [9]. Benefits of truncated multipliers include reduced area, delay, and power consumption [10]. An overview of truncated multipliers, which

$$
\begin{array}{cccccccccccccccc}
 & & & & & & A & & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & & & \times B & & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & & & & 1 & \overline{a_7b_0} & a_6b_0 & a_5b_0 & a_4b_0 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & & & & \overline{a_7b_1} & a_6b_1 & a_5b_1 & a_4b_1 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & & & & \overline{a_7b_2} & a_6b_2 & a_5b_2 & a_4b_2 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & & & & & \overline{a_7b_3} & a_6b_3 & a_5b_3 & a_4b_3 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
 & & & & \overline{a_7b_4} & a_6b_4 & a_5b_4 & a_4b_4 & a_3b_4 & a_2b_4 & a_1b_4 & a_0b_4 & & & & \\
 & & & \overline{a_7b_5} & a_6b_5 & a_5b_5 & a_4b_5 & a_3b_5 & a_2b_5 & a_1b_5 & a_0b_5 & & & & & \\
 & & \overline{a_7b_6} & a_6b_6 & a_5b_6 & a_4b_6 & a_3b_6 & a_2b_6 & a_1b_6 & a_0b_6 & & & & & & \\
1 & a_7b_7 & \overline{a_6b_7} & \overline{a_5b_7} & \overline{a_4b_7} & \overline{a_3b_7} & \overline{a_2b_7} & \overline{a_1b_7} & \overline{a_0b_7} & & & & & & & \\
\hline
p_{15} & p_{14} & p_{13} & p_{12} & p_{11} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Fig. 1. 8×8 partial product bit matrix (two's complement)

discusses several methods for correcting the error introduced due to unformed partial product bits, is given in [11]. The method used in this paper is constant correction, as described in [9].

Figure 2 shows an 8×8 truncated parallel multiplier with a correction constant added. The final result is l-bits long. We define k as the number of truncated columns that are formed, and r as the number of columns that are not formed. In this example, the five least significant columns of partial product bits are not formed ($l = 8$, $k = 3$, $r = 5$).

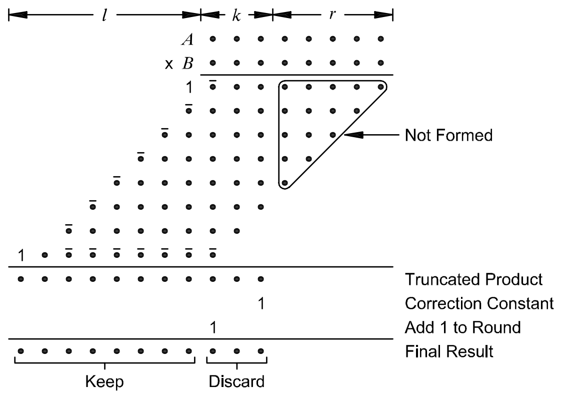


Fig. 2. 8×8 truncated multiplier with correction constant

Truncation saves an AND gate for each bit not formed and eliminates the full adders and half adders that would otherwise be required to reduce them to two rows. The delay due to reducing the partial product matrix is not improved because the height of the matrix is unchanged. However, a shorter carry propagate adder is required, which may improve the overall delay of the multiplier.

The correction constant, C_r, and the '1' added for rounding are normally included in the reduction matrix. In Figure 2 they are explicitly shown to make the concept more clear.

A consequence of truncation is that a reduction error is introduced due to the discarded bits. For simplicity, the operands are assumed to be integers, but the technique can also be applied to fractional or mixed number systems. With r unformed columns, the reduction error is

$$E_r = -\sum_{i=0}^{r-1} \sum_{j=0}^{i} a_{i-j} b_j 2^i \ . \tag{4}$$

If A and B are random with a uniform probability density, then the average value of each partial product bit is $\frac{1}{4}$, so the average reduction error is

$$E_{r\_avg} = -\frac{1}{4} \sum_{q=0}^{r-1} (q+1) 2^q = -\frac{1}{4} ((r-1) \cdot 2^r + 1) \ . \tag{5}$$

The correction constant, C_r, is chosen to offset E_{r_avg}. After rounding,

$$C_r = -\text{round}(2^{-r} E_{r\_avg}) \cdot 2^r = \text{round}\left((r-1) \cdot 2^{-2} + 2^{-(r+2)}\right) \cdot 2^r \ , \tag{6}$$

where round(x) indicates x is rounded to the nearest integer.

2 FIR Filter Architecture

This section describes the architecture used to study the effect of truncated multipliers in FIR filters. Little work has been published in this area, and this architecture incorporates the novel approach of combining all constants for two's complement multiplication and correction of reduction error into a single constant added just prior to computing the final filter output. This technique reduces the average reduction error of the filter by several orders of magnitude, when compared to the approach of including the constants directly in the multipliers. Section 2.1 presents an overview of the architecture, and Section 2.2 describes components within the architecture.

2.1 Architecture Overview

An FIR filter with T taps computes the following difference equation [12],

$$y[n] = \sum_{k=0}^{T-1} b[k] \cdot x[n-k] \ , \tag{7}$$

where $x[\,]$ is the input data stream, $b[k]$ is the k^{th} tap coefficient, and $y[\,]$ is the output data stream of the filter. Since the tap coefficients and the impulse response, $h[n]$, are related by

$$h[n] = \begin{cases} b[n], n = 0, 1, \ldots, T-1 \\ 0, \quad \text{otherwise,} \end{cases} \tag{8}$$

Equation (7) can be recognized as the discrete convolution of the input stream with the impulse response [12].

Figure 3 shows the block diagram of the FIR filter architecture used in this paper. This architecture has two data inputs, `x_in` and `coeff`, and one data output, `y_out`. There are two control inputs which are not shown, `clk` and `loadtap`.

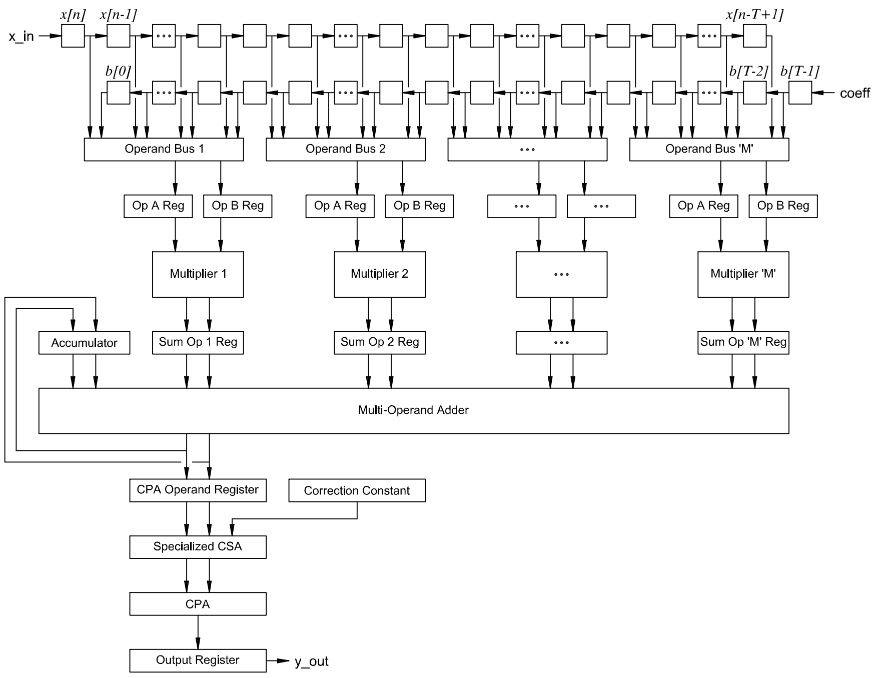


Fig. 3. Proposed FIR filter architecture with T taps and M multipliers

The input data stream enters at the `x_in` port. When the filter is ready to process a new sample, the data at `x_in` is clocked into the register labeled $x[n]$ in the block diagram. The $x[n]$ register is one of T shift registers, where T is the number of taps in the filter. When `x_in` is clocked into the $x[n]$ register, the values in the other registers are shifted right in the diagram, with the oldest value, $x[n-T+1]$ being discarded.

The tap coefficients are stored in another set of shift registers, labeled $b[0]$ through $b[T-1]$ in Figure 3. Coefficients are loaded into the registers by applying the coefficient values to the `coeff` port in sequence and cycling the `loadtap` signal to load each one.

The filter is pipelined with four stages: operand selection, multiplication, summation, and final addition.

Operand Selection: The number of multipliers in the architecture is configurable. For a filter with T taps and M multipliers, each multiplier performs

$\lceil T/M \rceil$ multiplications per input sample. The operands for each multiplier are selected each clock cycle by an operand bus and clocked into registers.

Multiplication: Each multiplier has two input operand registers, loaded by an operand bus in the previous stage. Each pair of operands is multiplied, and the final two rows of the reduction tree (the product in carry-save form) are clocked into a register where they become inputs to the multi-operand adder in the next stage. Keeping the result in carry-save form, rather than using a carry propagate adder (CPA), reduces the overall delay.

Summation: The multi-operand adder has carry-save inputs from each multiplier, as well as a carry-save input from the accumulator. After each of the $\lceil T/M \rceil$ multiplications have been performed, the output of the multi-operand adder (in carry-save form) is clocked into the CPA operand register where it is added in the next pipeline stage.

Final Addition: In the final stage, the carry-save vectors from the multi-operand adder and a correction constant are added by a specialized carry save adder and a carry propagate adder to produce a single result vector. The result is then clocked into an output register, which is connected to the `y_out` output port of the filter.

The `clk` signal clocks the system. The clock period is set so that the multipliers and the multi-operand adder can complete their operation within one clock cycle. Therefore, $\lceil T/M \rceil$ clock cycles are required to process each input sample. The final addition stage only needs to operate once per input sample, so it has $\lceil T/M \rceil$ clock cycles to complete its calculation and is generally not on the critical path.

2.2 Architecture Components

This section discusses the components of the FIR filter architecture.

Multipliers. In this paper, two's complement parallel tree multipliers are used to multiply the input data by the filter coefficients. When performing truncated multiplication, the constant correction method [9] is used. The output of each multiplier is the final two rows remaining after reduction of the partial product bits, which is the product in carry-save form [13]. Rounding does not occur at the multipliers, each product is $(l + k)$-bits long. Including the extra k bits in the summation avoids an accumulation of roundoff errors. Rounding is done in the final addition stage.

As described in Section 1.1, the last three terms in (3) are constants. In this architecture, these constants are *not* included in the partial product matrix. Likewise, if using truncated multipliers, the correction constant is not included either. Instead, the constants for each multiplication are added in a single operation in the final addition stage of the filter. This is described later in more detail.

Multi-operand Adder and Accumulator. As shown in (7), the output of an FIR filter is a sum of products. In this architecture, M products are computed per clock cycle. In each clock cycle, the carry-save outputs of each multiplier are added and stored in the accumulator register, also in carry-save form. The accumulator is included in the sum, except with the first group of products for a new input sample. This is accomplished by clearing the accumulator when the first group of products arrives at the input to the multi-operand adder.

The multi-operand adder is simply a counter reduction tree, similar to a counter reduction tree for a multiplier, except that it begins with operand bits from each input instead of a partial product bit matrix. The output of the multi-operand adder is the final two rows of bits remaining after reduction, which is the sum in carry-save form. This output is clocked into the accumulator register every clock cycle, and clocked into the CPA Operand Register every $\lceil T/M \rceil$ cycles.

Correction Constant Adder. As stated previously, the constants required for two's complement multipliers and the correction constant for unformed bits in truncated multipliers are not included in the reduction tree but are added during the final addition stage. A '1' for rounding the filter output is also added in this stage. All of these constants for each multiplier are precomputed and added as a single constant, C_{TOTAL}.

All multipliers used in this paper operate on two's complement operands. From (3), the constant which must be added for an $m \times n$ multiplier is $2^{m+n-1} + 2^{n-1} + 2^{m-1}$. With T taps, there are T multiply operations (assuming T is evenly divisible by M), so a value of

$$C_M = T(2^{m+n-1} + 2^{n-1} + 2^{m-1}) \tag{9}$$

must be added in the final addition stage.

The multipliers may be truncated with unformed columns of partial product bits. If there are unformed bits, the total average reduction error of the filter is $T \cdot E_{r\_avg}$. The correction for this is

$$C_R = \text{round}\left(T \cdot (r-1) \cdot 2^{-2} + T \cdot 2^{-(r+2)}\right) \cdot 2^r \quad . \tag{10}$$

To round the filter output to l bits, the rounding constant that must be used is

$$C_{RND} = 2^{r+k-1} \quad . \tag{11}$$

Combining these constants, the total correction constant for the filter is

$$C_{TOTAL} = C_M + C_R + C_{RND} \quad . \tag{12}$$

Adding C_{TOTAL} to the multi-operand adder output is done using a specialized carry-save adder (SCSA) which is simply a carry-save adder optimized for adding a constant bit vector. A carry-save adder uses full adders to reduce three bit vectors to two. SCSA's differ in that half adders are used in columns where the constant is a '0' and specialized half adders are used in columns where the constant is a '1'. A specialized half adder computes the sum and carry-out of two bits plus a '1', the logic equations being

$$s_i = \overline{a_i \oplus b_i} \qquad \text{and} \qquad c_{i+1} = a_i + b_i \ . \tag{13}$$

The output of the SCSA is then input to the final carry propagate adder.

Final Carry Propagate Adder. The output of the specialized carry-save adder is the filter output in carry-save form. A final carry propagate adder (CPA) is required to compute the final result. The final addition stage has $\lceil T/M \rceil$ clock cycles to complete, so for many applications a simple ripple-carry adder will be fast enough. If additional performance is required, a carry-lookahead adder may be used. Using a faster CPA does not increase throughput, but does improve latency.

Control. A filter with T taps and M multipliers requires $\lceil T/M \rceil$ clock cycles to process each input sample. The control circuit is a state machine with $\lceil T/M \rceil$ states, implemented using a modulo-$\lceil T/M \rceil$ counter. The present state is the output of the counter and is used to control which operands are selected by each operand bus. In addition to the present state, the control circuit generates four other signals: 1) `shiftData`, which shifts the input samples, 2) `clearAccum`, which clears the accumulator, 3) `loadCpaReg`, which loads the multi-operand adder output into the CPA operand register, and 4) `loadOutput`, which loads the final sum into the output register.

3 Filter Generation Software (FGS)

The architecture described in Section 2 provides a great deal of flexibility in terms of operand size, the number of taps, and the type of multipliers used. This implies that the design space is quite large. In order to facilitate the development of a large number of specific implementations, a tool was designed that automatically generates synthesizable structural VHDL models given a set of parameters. The tool, which is named FGS, also generates test benches and files of test vectors to verify the filter models.

FGS is written in Java and consists of two main packages. The arithmetic package, discussed in Section 3.1, is suitable for general use and is the foundation of FGS. The fgs package, discussed in Section 3.2, is specifically for generating the filters described previously. It uses the arithmetic package to generate the necessary components.

3.1 The arithmetic Package

The arithmetic package includes classes for modeling and simulating digital components. The simplest components include D flip-flops, half adders, and full adders. Larger components such as ripple-carry adders and parallel multipliers use the smaller components as building blocks. These components in turn are used to model complex systems such as FIR filters.

Common Classes and Interfaces. Figure 4 shows the classes and interfaces which are used by arithmetic subpackages. The most significant of these are VHDLGenerator, Parameterized, and Simulator.

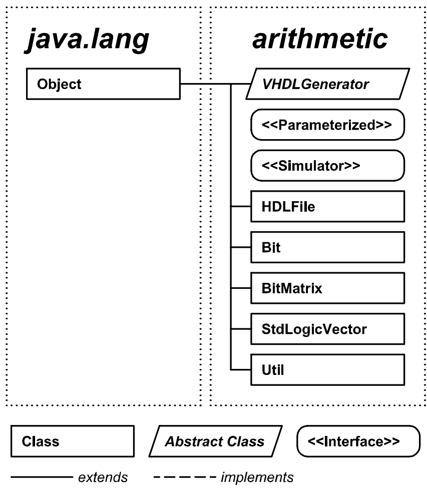


Fig. 4. The arithmetic package

VHDLGenerator is an abstract class. Any class that represents a digital component and can generate a VHDL model of itself is derived from this class. It defines three abstract methods which must be implemented by all subclasses. genCompleteVHDL() generates a complete VHDL file describing the component. This file includes synthesizable entity-architecture descriptions of all subcomponents used. genComponentDeclaration() generates the component declaration which must be included in the entity-architecture descriptions of other components which use this component. genEntityArchitecture() generates the entity-architecture description of this component.

Parameterized is an interface implemented by classes whose instances can be defined by a set of parameters. The interface includes get and set methods to access those parameters. Specific instances of Parameterized components can be easily modified by changing these parameters.

Simulator is an interface implemented by classes that can simulate their operation. The interface has only one method, simulate, which accepts a vector of inputs and returns a vector of outputs. These inputs and outputs are vectors of IEEE VHDL `std_logic_vectors` [14].

The arithmetic.smallcomponents Package. The arithmetic.smallcomponents package provides fundamental components including D flip-flops and full adders which are used as building blocks for larger components such as registers, adders, and multipliers. Each class in this package is derived from VHDLGenerator, enabling each to generate VHDL for use in larger components.

The arithmetic.adders Package. The classes in this package model various types of adders including carry propagate adders, specialized carry-save adders, and multi-operand adders. All components in these classes handle operands of arbitrary length and weight. This flexibility makes automatic VHDL generation more complex than it would be if operands were constrained to be the same length and weight. However, this flexibility is often required when an adder is used with another component such as a multiplier.

Figure 5 shows the arithmetic.adders package, which is typical of many of the arithmetic subpackages. CarryPropagateAdder is an abstract class from which carry propagate adders such as ripple-carry adders and carry-lookahead adders are derived. CarryPropagateAdder is a subclass of VHDLGenerator and implements the Simulator and Parameterized interfaces. Using interfaces and an inheritance hierarchy such as this help make FGS both straightforward to use and easy to extend. For example, a new type of carry propagate adder could be incorporated into existing complex models by subclassing CarryPropagateAdder.

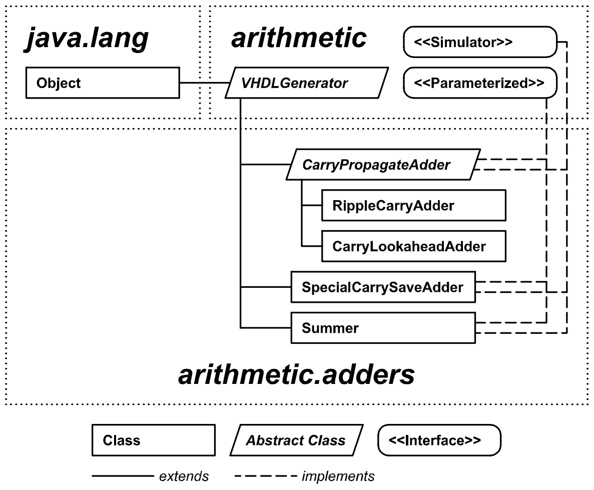


Fig. 5. The arithmetic.adders package

The arithmetic.matrixreduction Package. This package provides classes that perform matrix reduction, typically used by multi-operand adders and parallel multipliers. These classes perform Wallace, Dadda, and Reduced Area reduction [6–8]. Each of these classes are derived from the abstract class ReductionTree.

The arithmetic.multipliers Package. A ParallelMultiplier class was implemented for this paper and is representative of how FGS functions.

Parameters can be set to configure the multiplier for unsigned, two's complement, or combined operation. The number of unformed columns, if any, and the type of reduction, Wallace, Dadda, or Reduced Area, may also be specified. A BitMatrix object, which models the partial product matrix, is then instantiated and passed to a ReductionTree object for reduction. Through polymorphism (dynamic binding), the appropriate subclass of ReductionTree reduces the BitMatrix to two rows. These two rows can then be passed to a CarryPropagateAdder object for final addition, or in the case of the FIR filter architecture described in this paper, to a multi-operand adder.

The architecture of FGS makes it easy to change the bit matrix, reduction scheme, and final addition method. New techniques can be added seamlessly by subclassing appropriate abstract classes.

The arithmetic.misccomponents Package. This package includes classes that provide essential functionality but don't logically belong in other packages. This includes Bus, which models the operand busses of the FIR filter, and Register which models various types of data registers. Implementation of registers is done by changing the type of flip-flop objects which comprise the register.

The arithmetic.firfilters Package. This package includes classes for modeling ideal FIR filters as well as FIR filters based on the truncated architecture described in Section 2.

The "ideal" filters are ideal in the sense that the data and tap coefficients are double precision floating point. This is a reasonable approximation of infinite precision for most practical applications. The purpose of an ideal FIR filter object is to provide a baseline for comparison with practical FIR filters and allow measurement of calculation errors.

The FIRFilter class models FIR filters based on the architecture shown in Figure 3. All operands in FIRFilter objects are considered to be two's complement integers, and the multipliers and the multi-operand adder use Reduced Area reduction. There are many parameters that can be set including the tap coefficient and data lengths, the number of taps, the number of mulipliers, and the number of unformed columns in the multipliers.

The arithmetic.testing Package. This package provides classes for testing components generated by other classes, including parallel multipliers and FIR filters. The FIR filter test class generates a test bench and an input file of test vectors. It also generates a .vec file for simulation using Altera Max+Plus II.

The arithmetic.gui Package. This package provides graphical user interface (GUI) components for setting parameters and generating VHDL models for all of the larger components such as FIRFilter, ParallelMultiplier, etc. The GUI for each component is a Java Swing JPanel, which can be used in any Swing application. These panels make setting component parameters and generating VHDL files simple and convenient.

3.2 The fgs Package

Whereas the arithmetic package is suitable for general use, the fgs package is specific to the FIR filter architecture described in Section 2. fgs includes classes for automating much of the work done to analyze the use of truncated multipliers in FIR filters. For example, this package includes a driver class that automatically generates a large number of different FIR filter configurations for synthesis and testing. Complete VHDL models are then generated, as well as Tcl scripts to drive the synthesis tool. The Tcl script commands the synthesis program to write area and delay reports to disk files, which are are parsed by another class in the fgs package that summarizes the data and writes it to a CSV file for analysis by a spreadsheet application.

4 Results

Table 1 presents some representative synthesis results that were obtained from the Leonardo synthesis tool and the LCA300K 0.6 micron CMOS standard cell library. Additional data can be found in [15], which also also provides a more detailed analysis of the FIR filter architecture presented in this paper, including reduction and roundoff error. The main findings are:

1. Using truncated multipliers in FIR filters results in significant improvements in area. For example, the area of a 16-bit filter with 4 multipliers and 24 taps improves by 22.5 % with 12 unformed columns and by 36.4 % with 16 unformed columns. We estimate substantial power savings would be realized as well. Truncation has little impact on the overall delay of the filter.
2. The computational error introduced by truncation is tolerable for many applications. For example, the reduction error SNR for a 16-bit filter with 24 taps is 86.7 dB with 12 unformed columns and 61.2 dB with 16 unformed columns. In comparison, the roundoff error for an equivalent filter without truncation is 89.1 dB [15].
3. The average reduction error of a filter is independent of r (for $T > 4$), and much less than that of a single truncated multiplier. For a 16-bit filter with 24 taps and $r = 12$, the average reduction error is only 9.18×10^{-5} ulps, where an ulp is a unit of least precision in the 16-bit product. In comparison, the average reduction error of a single 16-bit multiplier with $r = 12$ is 1.56×10^{-2} ulps, and the average roundoff error of the same multiplier without truncation is 7.63×10^{-6} ulps.

Filter			**Synthesis Results**			**Improvement**			**Reduction Error**		
				Total	$A \cdot D$						
			Area	Delay	Product		Total	$A \cdot D$	$\mathrm{SNR}_{\mathrm{R}}$	σ_R	E_{AVG}
T	M	r	(gates)	(ns)	(gates·ns)	Area	Delay	Product	(dB)	(ulps)	(ulps)
12	2	0	16241	40.80	662633	—	—	—	∞	0	0
12	2	12	12437	40.68	505937	23.4%	0.3%	23.6%	89.70	0.268	-4.57E-5
12	2	16	10211	40.08	409257	37.1%	1.8%	38.2%	64.22	5.040	-4.57E-5
16	2	0	17369	54.40	944874	—	—	—	∞	0	0
16	2	12	13529	54.24	733813	22.1%	0.3%	22.3%	88.45	0.310	-6.10E-5
16	2	16	11303	53.44	604032	34.9%	1.8%	36.1%	62.97	5.820	-6.10E-5
20	2	0	19278	68.00	1310904	—	—	—	∞	0	0
20	2	12	15475	67.80	1049205	19.7%	0.3%	20.0%	87.48	0.346	-7.60E-5
20	2	16	13249	66.80	885033	31.3%	1.8%	32.5%	62.00	6.508	-7.60E-5
24	2	0	20828	81.60	1699565	—	—	—	∞	0	0
24	2	12	17007	81.36	1383690	18.3%	0.3%	18.6%	86.69	0.379	-9.18E-5
24	2	16	14781	80.16	1184845	29.0%	1.8%	30.3%	61.21	7.143	-9.18E-5
12	4	0	25355	20.40	517242	—	—	—	∞	0	0
12	4	12	18671	20.34	379768	26.4%	0.3%	26.6%	89.70	0.268	-4.57E-5
12	4	16	14521	20.04	291001	42.7%	1.8%	43.7%	64.22	5.040	-4.57E-5
16	4	0	26133	27.20	710818	—	—	—	∞	0	0
16	4	12	19413	27.12	526481	25.7%	0.3%	25.9%	88.45	0.310	-6.10E-5
16	4	16	15264	26.72	407854	41.6%	1.8%	42.6%	62.97	5.820	-6.10E-5
20	4	0	28468	34.00	967912	—	—	—	∞	0	0
20	4	12	21786	33.90	738545	23.5%	0.3%	23.7%	87.48	0.346	-7.60E-5
20	4	16	17636	33.40	589042	38.0%	1.8%	39.1%	62.00	6.508	-7.60E-5
24	4	0	29802	40.80	1215922	—	—	—	∞	0	0
24	4	12	23101	40.68	939749	22.5%	0.3%	22.7%	86.69	0.379	-9.18E-5
24	4	16	18950	40.08	759516	36.4%	1.8%	37.5%	61.21	7.143	-9.18E-5

Table 1. Synthesis results for 16-bit operands, output rounded to 16-bits (optimized for area)

5 Conclusions

This paper presents a tool used to rapidly prototype parameterized FIR filters. The tool is used to study the effects of using truncated multipliers in those filters. It is based on a package of arithmetic classes that are used as components in hierarchical designs, and are capable of generating structural level VHDL models of themselves. Using these classes as building blocks, FirFilter objects generate complete VHDL models of specific FIR filters. The arithmetic package is extendable and suitable for use in other applications, enabling rapid prototyping of other computational systems. As a part of ongoing research at Lehigh University, the tool is being expanded to study other DSP applications, and will be made available to the public in the near future.

References

1. Lightbody, G., Walke, R., Woods, R.F., McCanny, J.V.: Rapid System Prototyping of a Single Chip Adaptive Beamformer. (In: Proceedings of Signal Processing Systems) 285–294
2. McCanny, J., Ridge, D., Yi, H., Hunter, J.: Hierarchical VHDL Libraries for DSP ASIC Design. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing. (1997) 675–678
3. Pihl, J., Aas, E.J.: A Multiplier and Squarer Generator for High Performance DSP Applications. In: Proceedings of the 39th Midwest Symposium on Circuits and Systems. (1996) 109–112
4. Richards, M.A., Gradient, A.J., Frank, G.A.: Rapid Prototyping of Application Specific Signal Processors. Kluwer Academic Publishers (1997)
5. Saultz, J.E.: Rapid Prototyping of Application-Specific Signal Processors (RASSP) In-Progress Report. Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology (1997) 29–47
6. Wallace, C.S.: A Suggestion for a Fast Multiplier. IEEE Transactions on Electronic Computers **EC-13** (1964) 14–17
7. Dadda, L.: Some Schemes for Parallel Multipliers. Alta Frequenza **34** (1965) 349–356
8. Bickerstaff, K.C., Schulte, M.J., Swartzlander, Jr., E.E.: Parallel Reduced Area Multipliers. IEEE Journal of VLSI Signal Processing **9** (1995) 181–191
9. Schulte, M.J., Swartzlander, Jr., E.E.: Truncated Multiplication with Correction Constant. In: VLSI Signal Processing VI, Eindhoven, Netherlands, IEEE Press (1993) 388–396
10. Schulte, M.J., Stine, J.E., Jansen, J.G.: Reduced Power Dissipation Through Truncated Multiplication. In: IEEE Alessandro Volta Memorial Workshop on Low Power Design, Como, Italy (1999) 61–69
11. Swartzlander, Jr., E.E.: Truncated Multiplication with Approximate Rounding. In: Proceedings of the 33rd Asilomar Conference on Signals, Circuits, and Systems. (1999) 1480–1483
12. Oppenheim, A.V., Schafer, R.W.: Discrete-Time Signal Processing, 2nd edition. Prentice Hall, Upper Saddle River, NJ (1999)
13. Koren, I.: Computer Arithmetic and Algorithms. Prentice Hall, Englewood Cliffs, NJ (1993)

14. : IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std-logic1164): IEEE Std 1164-1993 (26 May 1993)
15. Walters III, E.G.: Design Tradeoffs Using Truncated Multipliers in FIR Filter Implementations. Master's thesis, Lehigh University (2002)