Parallelism Utilization in Embedded Reconfigurable Computing Systems: A Survey of Recent Trends

S. Arash Ostadzadeh and Koen Bertels
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands
{arash, koen}@ce.et.tudelft.nl

Abstract— **Recently, embedded reconfigurable computing has attracted great attention due to its potential to accelerate application execution. Its key feature is the ability to perform computations in hardware to increase performance, while retaining much of the flexibility of a software solution. Researchers in this field have reported substantial performance improvements for a variety of different applications like cryptography, multimedia processing, genetics, networking and DSP. Embedded reconfigurable computing systems can lend themselves to high performance computing by taking advantage of parallelism at different levels of granularity, ranging from fine grained instruction level to coarse grained process and/or task level parallelism. It is often necessary to use different parallel processing techniques to fully take advantage of these systems. In this survey, we explore recent enhancements to this new field of computing, considering the embedded reconfigurable hardware architectures and software facilities targeting these systems. The focus of the survey is on the employment of parallelism which seems to be a key feature in application development for embedded reconfigurable systems. More precisely, four different levels of parallelism indicated by Instruction Level, Data/Loop Level, Task Level, and Process/Thread Level, are introduced and distinguished by properties identified for each category. Various reconfigurable systems incorporating one or a combination of these attributes are investigated. Finally, we generally try to identify the major problems that limit the embedded reconfigurable computing systems from reaching their maximum potentials.**

Keywords: **Embedded Reconfigurable Computing, FPGA, Parallelism, Hardware-Software Co-design, Design/Performance Evaluation**

I. Introduction

Tuning applications for optimum performance has always been elusive. Recently, Reconfigurable Computing (RC) has attracted great attention because of its potential to accelerate application execution. Its key feature is the ability to perform computations in hardware to increase performance, while retaining much of the flexibility of a software solution [1][2].

Traditionally, two methods are used in implementing algorithms. The first one is with the hardwired technology which can use an Application Specific Integrated Circuit (ASIC) or separate components combined together on board(s). ASICs are dedicated components designed for specific computations, and thus they prove to be quite fast and efficient when executing the exact computation for which they were designed. A more flexible method would be the employment of software-programmable microprocessors in which processors execute a set of instructions to perform computations. Simply changing the software instructions would result in the system functionality alternation without any hardware manipulation. However, this flexibility is going to be compensated by degradation in performance [1].

Basically, the dynamic nature of reconfigurable devices is addressed to a collection of computational elements or logic blocks whose functionality is decided by means of configuration patterns. The logic blocks are composed of either commercial FPGAs or custom configurable hardware. This new field of computing has attracted great attention and shown to be efficient in a variety of applications to name a few, network intrusion prevention [3], data encryption [4][5][6], image processing [7], digital signal processing [8][9][10][11], and recently more sophisticated applications like face recognition [12].

Reconfigurable systems are usually employing general-purpose processor(s) (GPP) in addition to logic blocks to exhibit better flexibility in executing wide range of applications and achieve better performances. These GPPs usually conduct hard-to-map kernels (codes) which can be easily implemented in software such as control constructs, while the reconfigurable units are mostly concerned with computations.

Compilation environments for reconfigurable device are quite diverse, ranging from tools for hand-mapping assistance of a digital unit to the hardware, to fully automated utilities which are fed with circuit description in a high-level language such as VHDL to be transformed into a configuration for reconfigurable device [13].

It should also be noted that for reconfigurable devices which demand fixed structure setting at compile-time, space availability can be a source of concern and the system can only accelerate as much of the program as fits in the start-up, however, for the run-time reconfigurable devices, it's possible to reuse the space during program execution which is going to introduce a tradeoff between extra reconfigurability time and increased efficiency arising from the potential acceleration of larger part of the program. Although reconfigurable systems (even run-time ones) have been generally shown to achieve high performance, the speedups over traditional microprocessor systems are limited by the cost of configuration of the hardware. Current re-

configurable systems suffer from a significant overhead due to the time it takes to reconfigure their hardware. In order to deal with this overhead, and increase the computing power of reconfigurable systems, it is important to develop hardware and software systems to reduce or eliminate this delay. To reduce the overhead of configuration, some techniques like configuration compression [14][15], configuration prefetching technique [16] and partial reuse of already programmed configurations [17][18][19][20] are employed.

An introduction to the rapidly evolving field of reconfigurable computing, background of techniques and systems, current researches in hardware and software for RC, and techniques for run-time reconfigurability can be found in [21][22][23][24][1][25][26]. Several commercial dynamically reconfigurable processors systems and their array structures, PEs and interconnection architectures are classified in [27].

The rest of this paper is organized as follows. In section II, we introduce parallelism utilization at different levels of granularity in recent works on embedded reconfigurable systems. In section III, the modern concept of High-Performance Reconfigurable Computing is concisely described. Some limiting factors in current embedded systems are discussed in section IV. Finally, section V is devoted to the summary and concluding remarks. It should also be stressed that the paper primarily presents an outline of only the most recent developments and techniques in the reconfigurable computing field and is not intended as an introductory text or detailed descriptions of ongoing research projects.

II. Parallel Computing With Reconfigurable Logic

Embedded reconfigurable computing is bound to exploit parallelism at different levels of granularity. In this section, we are going to take a brief look at the reconfigurable devices' roles and contributions in this context. Regarding the standard parallel computing terminology, it is ostensible that reconfigurable hardware can generally provide parallelism in finer granularity as compared to distributed systems, up to the task or process level. Upper level coarse-grain parallelism is commonly practiced on distributed systems in an abstract layer over the reconfigurable logic.

In the following, we first introduce different levels of parallelism and quickly survey the use of embedded reconfigurable systems at various granularity of parallelism, ranging from instruction through process level. Figure 1 depicts the concept of parallelism at different levels in embedded reconfigurable computing systems.

A. Instruction Level Parallelism (ILP)

The lowest level of granularity is ILP and is exploited by high-performance microprocessors. Instruction level parallelism in its purest form is truly at the instruction level, such as load, stores, and ALU

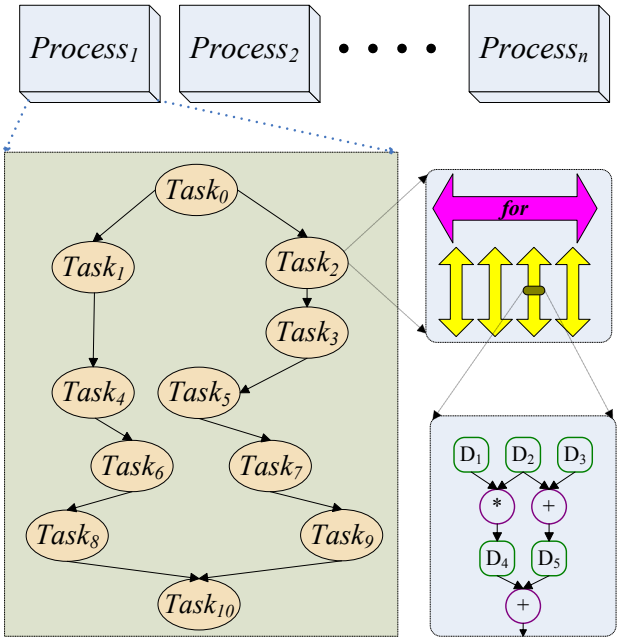

Fig. 1. Different levels of parallelism in Embedded Reconfigurable Computing Systems

operations. The instruction level operations are not visible in the source code and so cannot be manipulated by the programmer. In conventional microprocessors, ILP is exploited in the micro-architecture of a superscalar processor. By having multiple instructions in progress in different stages of completion, the superscalar processor is able to complete more than one instruction in a clock cycle. Very Long Instruction Word (VLIW) processors offer another method for fine-grained parallel operation. A VLIW processor contains multiple functional units operating in parallel.

The key difference between VLIWs and SIMDs is the instruction stream. In a VLIW of P processing elements, there can be P different instructions executed whereas a SIMD executes the same instruction across all processing elements. The parallelism of a VLIW, however, is limited by the scalability of the register file, which is shared by all processing elements, and is further limited by the instruction level parallelism of the source code [28]. VLIW processors have started gaining acceptance in the embedded systems domain. However, monolithic register file VLIW processors with a large number of functional units are not practical. A simple solution would be the utilization of architectures termed Clustered VLIW Processors [29].

RC-SIMD [30] is a type of SIMD architecture, with a reconfigurable communication network. It uses a delay-line in the instruction bus, causing the accesses to the communication network to be distributed over time. This architecture requires only a very cheap

communication network while performing almost the same as expensive fully connected SIMD architectures. A conflict model is also employed to deal with the irregular resource conflicts in this architecture. A runtime reconfiguration refinement of the SIMD architecture is presented in [31].

A multiprocessor platform for high throughput decoding based on a configurable ASIP combined with an efficient memory and communication interconnect scheme is presented in [32]. This ASIP has an SIMD architecture with a specialized and extensible instruction-set and 5-stages pipeline control. The attached memories and communication interfaces enable the design of efficient multiprocessor architectures. These multiprocessor architectures benefit from the recent shuffling technique introduced in the turbo-decoding field to reduce communication latency. The major characteristics of the proposed platform can be addressed to its flexibility and scalability which make it reusable for various operating modes.

Bocchino and Adve [33] have presented Vector LLVA, a virtual instruction set architecture (VISA) that exposes extensive static information about vector parallelism while avoiding the use of hardware-specific parameters. It provides both arbitrary-length vectors and fixed-length vectors, such as subword SIMD extensions, together with a set of operations on both vector types. Translators that compile (1) Vector LLVA written with arbitrary-length vectors to the Motorola RSVP architecture and (2) Vector LLVA written with fixed-length vectors to both AltiVec and Intel SSE2 are implemented. The experiments show that VISA design captures vector parallelism for two quite different classes of architectures and provides virtual object code portability within the class of subword SIMD architectures.

Some modern microprocessors use an out-of-order execution mechanism to keep multiple execution units as busy as possible and achieve a higher performance. This is achieved by allowing instructions to be issued and completed out of the original program sequence as a means of exposing concurrency in a sequential instruction stream. More than one instruction can be issued in each cycle, but only independent instructions can be executed in parallel, other instructions must be kept waiting or, under some circumstances, can proceed speculatively. Speculative execution and out-of-order issue are used in superscalar processors to expose concurrency from sequential binary code. The Sun MAJC 5200 is a chip multiprocessors based on four-way issue, VLIW pipelines. This architecture provides a set of predicated instructions to support control speculation. Intel's explicitly parallel instruction computing (EPIC) architecture is another speculative evolution of VLIW.

To compile for a superscalar processor, the compiler simply generates a sequential instruction stream, and the processor parallelizes the instruction stream at run time. In contrast, the VLIW processor executes the instruction word generated by the compiler, requiring the compiler to schedule concurrent operations at compile time. Some legacy techniques in compiling sequential code for reconfigurable processors concerning ILP are reviewed in [34].

Explicit Dataflow Graph Execution (EDGE) architectures [35] aim to exploit fine-grained concurrency within a single thread. They break a program into a sequence of multi-instruction blocks that must each commit atomically. In the TRIPS prototype EDGE architecture, the compiler assigns instruction numbers that determine placement on the ALU substrate. To exploit instruction level parallelism, the TRIPS microarchitecture implements out-of-order execution. By assigning IDs to instructions, the TRIPS scheduler statically places each instruction on the array of ALUs, and the hardware dynamically issues instructions when their operands are ready. It differs from the VLIW approach, which uses static placement and static issue, and the out-of-order superscalar approach, which uses dynamic placement and dynamic issue.

VLIW-like architectures have large hardware resources supporting multiple parallel operations. Therefore, performance improvements can easily be obtained by defining and using a complex instruction with multiple parallel operations instead of using a sequence of simple instructions, however, a typical RISC architecture has little instruction level parallelism and thus cannot benefit from parallel operations combined into one complex instruction. However, even in RISC architectures with no explicit instruction level parallelism, we can exploit the parallelism in between the pipeline stages. Many other possibilities that combine multiple operations from different pipeline stages are also feasible, and can contribute to performance improvements over the processor's native instruction set. Lee et al. [36] have presented an instruction set synthesis framework that optimizes the instruction set through an efficient instruction encoding for the given application as well as for the given data path architecture. Their work is aimed at modern RISC pipelined architectures with multi-cycle instruction support, representative of current configurable processors despite most existing methodologies [37][38] which are only applicable to VLIW-like processors. A library of new instructions is created with various encoding alternatives taking into account the data path architecture constraints, and then the best set of instructions is selected while satisfying the instruction bitwidth constraint.

Paolucci et al. [39] have proposed a tiled architectural strategy that employs scalable building blocks. In SHAPES [40], a typical DSP oriented tile is composed of a RISC, a VLIW DSP, a DNP (Distributed Network Processor), on-tile memories and a set of on-

tile peripherals (POT). Each tile can be equipped with a Distributed eXternal Memory (DXM). The tile is the evolution of Atmel Diopsis, a multiprocessor SoC which includes a RISC plus a floating-point VLIW mAgic DSP. The SHAPES routing fabric connects on-chip and off-chip tiles, weaving a distributed packet switching network. SHAPES will investigate a layered system software, which does not destroy algorithmic and distribution info provided by the programmer and is fully aware of the hardware paradigm. In [41], a parameterizable hierarchical instruction scheduling for tiled processors is explored. The scheduler is employed to determine the contention-latency sweet spot that generates the best instruction schedule for each application. To avoid the application-specific tuning, the parameters that produce the best performance across all applications are determined.

Adaptive Explicitly Parallel Instruction Computing (AEPIC) [42]is a stylized form of a reconfigurable system-on-a-chip that is designed to enable compiler control of reconfigurable resources. AEPIC is an architectural model that extends the Explicitly Parallel Instruction Computing Architectures (EPIC) [43] design outline, wherein instruction level parallelism is explicitly communicated by the compiler to the hardware. AEPIC extends this model into reconfigurable SoCs. It has support for a reconfigurable fabric, which can be dynamically configured as a collection of Adaptable Functional Units (AFUs). The AFUs are configured by a stream of bits called a configuration. AEPIC specific instructions are executed on the EPIC processor to send the configuration stream to the adaptive fabric. Similar instructions are executed to manage configurations within the configuration memory hierarchy.

2D-VLIW [44] is another architecture and execution model which adopts a template based on large pieces of computation running over a matrix of functional units connected by a set of local register spread across the matrix. Experiments show performance gain when comparing to an EPIC architecture with the same number of registers and functional units.

Jones et al. [28] have proposed a Super-Complex Instruction-Set Computing (SuperCISC) embedded processor architecture and particularly investigated the performance and power consumption of this device compared to traditional processor architecture-based execution. SuperCISC is a heterogeneous, multicore processor architecture designed to exceed performance of traditional embedded processors while maintaining a reduced power budget compared to low-power embedded processors. These processors are tightly integrated through a shared register file, eliminating the need for overheads associated with busses often used in multiprocessor chips. At the heart of the SuperCISC processor is a multicore VLIW containing several homogeneous execution cores/functional units.

Code generation for embedded reconfigurable architectures can efficiently exploit multiple memory banks to demonstrate instruction level parallelism. Basically, such code generation process should contain intermediate representation, code compaction, instruction scheduling, memory bank assignment, and register/accumulator assignment [45]. These five phases can be performed in various sequences because they are logically independent. A number of researches have investigated the use of multi-bank memory to achieve maximum instruction level parallelism. Some works focus on designing variable partitioning mechanisms, which try to evenly distribute memory accesses and explore the potential of higher memory bandwidth [46][47]. For heterogeneous register sets, Daveau et al. [48] and Zhuang et al. [49] present specific register allocation algorithms to fit their irregularities. Detailed information is given in [50].

B. Data Level Parallelism (DLP)

Data level parallelism which sometimes referred to as Loop Level Parallelism (LLP), is usually revealed and exploited in sequential programs by optimizing compilers. Data parallelism on a small scale can be found in basic blocks of a program. Larger scale data parallelism can be found in a nest of loops that perform array/vector calculations.

Loop parallelization is a common technique in embedded reconfigurable computing as the primary target to exploit parallelism. Generally, automatic parallelization of loops is planned by hardware compilers as an attempt to maximize the use of the reconfigurable hardware. Loop unrolling can be done in different styles and at various scales. Some compilers select the innermost loop level to be completely unrolled for parallel execution in hardware, potentially creating a heavily pipelined structure [51][52]. For these cases, outer loops may not have multiple iterations executing simultaneously.

Any loop reordering to improve the parallelism utilization of the reconfigurable device is basically done by the programmer as a preprocessing step. On the other hand, some compiler systems have taken this procedure a step further and focus on the parallelization of all loops within the program, not just the inner loops [53][54]. These type of compilers generate and analyze different data and control flow graph representations based upon the entire program source code as needed to clear data and control dependencies between iterations and schedule parallel operations in the hardware. Kejariwal et al. [55] have evaluated the performance potential of different types of parallelism when executing loops. Applications from the industry-standard EEMBC 1.1, EEMBC 2.0 and the MiBench embedded benchmark suites are analyzed using the Intel C compiler.

Ziegler and Hall [56] have presented a set of measurements which characterize the design space for auto-

matically mapping high-level algorithms consisting of multiple loop nests onto an FPGA. They have focused on the space-time tradeoffs associated with sharing constrained chip area among multiple computations represented by an asynchronous pipeline. Other analyses and transformations, also associated with parallelizing compiler technology, are used to perform high-level optimization of the designs. The amount of parallelism in individual loop nests is controlled variably with the goal of deriving an overall design that makes the most effective use of chip resources. They have described several heuristics for automatically searching the space and a set of metrics for evaluating and comparing designs.

Traditional schedulers for mapping software implementations of compute-intensive loops onto the array can be less profitable if they do not take into account the explicit routing of operand values. In essence, the problem of binding operations to time slots and resources is extended to also include explicit routing of operands from producers to consumers. A software pipelining technique referred to as modulo graph embedding [57] is proposed for mapping loop bodies onto reconfigurable architectures. It leverages graph embedding from graph theory, which is used to draw graphs onto a target space. The advantage of the technique is that it considers the communication structure of the loop body during mapping.

Dou et al. [58] have also presented a speculative execution mechanism for dynamic loop scheduling with the goal of one iteration per cycle. Their technique exploits both data dependences of intra-iteration and inter-iteration. Two instructions for special data reuses in the case of loop-carried dependences have been designed. The experimental results show substantial reduction in memory accesses.

C. Task Level Parallelism (TLP)

Task level parallelism is usually regarded as the first potential of parallelism brought into attention in many applications which can result in considerable performance gains. It is particularly important in embedded reconfigurable systems because these systems often perform several different types of computation on data streams. Conceptually, task level parallelism looks easy to exploit since tasks can be allocated to processors. However, task structure is not easily exposed in ordinary programming languages; programmers often face difficulties to clarify the bounds of tasks. More abstract programming models may help clarify the task structure of an application.

Embedded processes can also consist of multiple (sub)tasks that are presented as different source code which can be (partly) executed in parallel. However, the subtask level parallelism inside a single task is often too limited to fully utilize all the parallel processors and results in many slacks on processors. Subtasks of multiple tasks can also be executed in an interleaving fashion for more efficient use of the processors. In [59] design-time algorithms are proposed to interleave subtasks based on the separated schedules of tasks. The interleaver can be considered as part of a hierarchical scheduler to steer the code generation of very complex applications with many tasks.

Krasteva et al. [60] have proposed an FPGA partition architecture, a methodology and a set of supporting tools that enable the use of partial reconfiguration in two directions: the (re)allocation of tasks within a slot based FPGA arrangement, and the reconfiguration of the communication infrastructure between these tasks and with an external processor. Thus, embedded reconfigurable devices can operate autonomously to adapt themselves when they receive a new task or group of tasks, optimizing both task allocation and intra-task communications. The type of communication structures supported can be a combination of buses, point-to point connections and networks-on-chip (NoC), each with variable width, sharing a fixed set of intra-task communication channels.

D. Process vs. Thread Level Parallelism

In parallel computing systems, the major parallel activity can be at a process level or the finer granularity thread level. At process level, each process has its own separate address space. In order to communicate state, a process must send a message and the destination process(es) must explicitly receive the message. Several different sorts of messaging protocols may be used. Messages may be buffered, asynchronous, as with MPI [61]. At the thread level of parallel processing, the threads share an address space, and can communicate through shared memory or messaging. Signaling and synchronization mechanisms for thread-based processing include critical sections and mutual exclusion, or barriers.

A multithreaded processor is able to pursue two or more threads of control in parallel within the processor pipeline. The contexts of two or more threads of control are often stored in separate on-chip register sets. Unused instruction slots, which arise from latencies during the pipelined execution of single-threaded programs by a contemporary microprocessor, are filled by instructions of other threads within a multithreaded processor. The execution units are multiplexed between the thread contexts that are loaded in the register sets. Underutilization of a superscalar processor due to missing ILP can be overcome by simultaneous multithreading, where a processor can issue multiple instructions from multiple threads each cycle. Simultaneous multithreaded processors combine the multithreading technique with a wide-issue superscalar processor to utilize a larger part of the issue bandwidth by issuing instructions from different threads simulta-

neously.

Explicit multithreaded processors are multithreaded processors that apply processes or operating system threads in their hardware thread slots. These processors optimize the throughput of multiprogramming workloads rather than single-thread performance. We distinguish these processors from implicit multithreaded processors that utilize thread-level speculation by speculatively executing compiler- or machine-generated threads of control that are part of a single sequential program. Many forms of explicit multi-threading techniques have been described, such as interleaved multi-threading (IMT), blocked multithreading (BMT) and SMT.

Simultaneous multithreading (SMT) is an architectural technique that improves resource utilization by allowing instructions from multiple threads to coexist in a processor and share resources, however, it cannot be afforded in systems with tight energy budgets. Moreover, it does not exploit data level parallel hardware, and does not exploit the available information on threads. Earlier studies have shown that the performance of an SMT architecture begins to saturate as the number of coexisting threads increases beyond four. Shin et al. [62] have shown that no single fetch policy can be the best solution during the entire execution time and that a significant performance improvement can be attained by dynamically switching the fetch policies. They proposed an implementation method which includes an extremely lightweight thread to control fetch policies (a detector thread) and a processor architecture to run the detector thread without impact on the user application threads. They have also evaluated various heuristics for the detector thread to determine the best fetch policies. Software-SMT (SW-SMT) [63], is another technique to exploit task level parallelism to improve the utilization of both instruction level and data level parallel hardware, thereby improving performance. The technique performs simultaneous compilation of multiple threads at design-time, and it includes a run-time selection of the most efficient mixes. A good survey of multithreading and classifications of explicit multithreading techniques is given in [64].

An alternative, which avoids complexity in instruction issue and elimination of speculative execution, is the microthreaded model. This model fragments sequential code at compile time and executes the fragments out of order while maintaining in-order execution within the fragments. The only constraints on the execution of fragments are the dependencies between them, which are managed in a distributed and scalable manner using synchronizing registers. The fragments of code are called microthreads and they capture instruction level parallelism [65].

Carta et al. [66] have proposed a reconfigurable architectural template which exploits mixed coarse-grained and fine-grained reconfigurable data path and control elements to obtain performances at ASICs level on computational tasks based on repetitive execution of a reduced set of operations on multidimensional array data. The architectural template determines execution partitioning between dominant and non-dominant kernels, and processor/coprocessor interaction. System integrability and scalability is improved by the use of memory mapping for coprocessor/processor communications. In this way coprocessor can be coupled with each kind of existing processor, and a cluster of coprocessors in parallel can be used to implement thread-level parallelism. The main innovation lies in reconfigurable coprocessor core architecture which is optimized for the execution of the kernels.

ALP [67] is also an architecture that efficiently integrates all different forms of parallelisms (including ILP and TLP) with evolutionary changes to programming model and hardware. The novel part of ALP is a data level parallelism technique called SIMD vectors and streams (SVectors/SStreams), which is integrated within a conventional superscalar-based CMP/SMT architecture with subword SIMD. This technique lies between subword SIMD and vectors, providing significant benefits over the former at a lower cost than the latter.

ATLAS [68], is a prototype for chip-multiprocessors with hardware support for Transactional Memory (TM), a technology aiming to simplify parallel programming. ATLAS uses BEE2 multi-FPGA board to provide a system with 8 PowerPC cores running Linux. It's aimed to provide benefits for chip-multiprocessors research such as performance improvement over a software simulator and good visibility that helps with software tuning and architectural enhancements. Certain issues about building a FPGA-based framework for chip-multiprocessors are also addressed in the work, such as overall performance, challenges of mapping ASIC-style chip-multiprocessors RTL on to FPGAs, software support, the selection criteria for the base processor, and the challenges of using pre-designed IP libraries.

Currently researchers in MIT are investigating vector-thread architectures. These architectures unify data level, thread level, and instruction level parallelism, providing new insights of parallelization for programs that are difficult to vectorize or that incur excessive synchronization costs when multithreaded. They have developed the Scale processor, which is an example of a vector-thread architecture designed for low-power and high-performance embedded systems. The prototype includes a single-issue 32-bit RISC control processor, a vector-thread unit which supports up to 128 virtual processor threads and can execute up to 16 instructions per cycle, and a 32KB shared primary cache [69].

The design of a thread-associative memory microar-

chitecture for multicore and multithreaded processor is investigated in [70]. Considering the fact that memory contention among concurrent threads in chip multithreaded processing has become a limiting factor for performance improvement, the proposed thread-associative memory addresses this challenge by incorporating thread-specific information explicitly into on-chip memory hardware and can be utilized at different levels of memory hierarchy.

Coarse-grained reconfigurable architecture ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) and its compiler offer high instruction level parallelism to applications by means of a sparsely interconnected array of functional units and register files. Wu et al. [71] have proposed ADRES extension to MT-ADRES (Multi-Threaded ADRES) to also exploit thread level parallelism. On MT-ADRES architectures, the array can be partitioned in multiple smaller arrays that can execute threads in parallel. Because the partition can be changed dynamically, this extension provides more flexibility than a multi-core approach.

Anderson et al. [72] have presented hthreads, a unifying programming model for specifying application threads running within a hybrid CPU/FPGA system. Threads are specified from a single pthreads multithreaded application program and compiled to run on the CPU or synthesized to run on the FPGA. The hthreads system abstracts the CPU/FPGA components into a unified custom threaded multiprocessor architecture platform. To support the abstraction of the CPU/FPGA component boundary, they have created the hardware thread interface (HWTI) component that frees the designer from having to specify and embed platform specific instructions to form customized hardware/software interactions. Instead, the hardware thread interface supports the generalized pthreads API semantics, and allows passing of abstract data types between hardware and software threads. Thus the hardware thread interface provides an abstract, platform independent compilation target that enables thread and instruction level parallelism across the software/hardware boundary.

Fuentes [73] have proposed a lightweight subset implementation of the MPI standard, called TMD-MPI. TMD-MPI provides a programming model capable of using multiple-FPGAs and embedded processors while hiding hardware complexities from the programmer, facilitating the development of parallel code and promoting code portability. A message-passing engine (TMD-MPE) is also developed to encapsulate the TMD-MPI functionality in hardware. TMD-MPE enables the communication between hardware engines and embedded processors. In addition, a Network-on-Chip is designed to enable intra-FPGA and inter-FPGA communications. Together, TMD-MPI, TMD-MPE and the network provide a flexible design flow for Multiprocessor System-on-Chip design.

III. High-Performance Reconfigurable Computing (HPRC)

High-Performance Reconfigurable Computing is based on uniting the conventional processors and field-programmable gate arrays. This new powerful computing facility is basically looks appealing for the high-performance computing community, primarily because these systems have the potential to exploit coarse-grained process/task level parallelism as well as fine-grained instruction level parallelism through direct hardware execution on FPGAs and dynamically tune their architecture to fit various applications [25][74]. Figure 2 illustrates the outline of a typical high-performance reconfigurable architecture.

High-performance reconfigurable computers, also known as reconfigurable supercomputers, have shown considerable improvements not only in performance but also power, size, and cost over conventional high-performance computers. Traditionally HPRC benefits are utilized in compute-intensive integer applications. However, there has been doubt that the same benefits can be attained for general scientific applications. Programming HPRCs is also not straightforward yet and depending on the programming tool can range from designing hardware to software programming which may require partial or substantial hardware knowledge. Fortunately, the trend in reconfigurable chip sizes and diversity of resources may relieve some of those concerns. Yet, with the hardware reconfigurability, it is feared that users have to learn how to design hardware if they were to employ such machines effectively [75].

The recent SRC-6 and SRC-7 parallel architectures from SRC Computers are developed with a gear toward scalability [76]. In addition, traditional high-performance computing vendors have utilized FPGAs into their parallel architectures. Silicon Graphics Inc. (SGI) has introduced SGI RASC (Reconfigurable Application Specific Computing) technology [77]. Building on the success of the Cray XT3 system, the Cray XT4 system brings new levels of scalability and sustained performance to high performance computing [78]. Linux Networx [79] is working on an FPGA-accelerated system tailored specifically for high performance multi-paradigm computing. It aims to help accelerate adoption of this emerging computing paradigm by offering a powerful platform with open-source tools. In keeping with the Linux spirit, the driver, user-space API, and Verilog interfacing IP will all be open-source and transparent. It's claimed that the system would be the most powerful reconfigurable device available on the market.

As far as software development is concerned, SRC provides a semi-integrated solution that addresses the hardware and software sides of the application sepa-

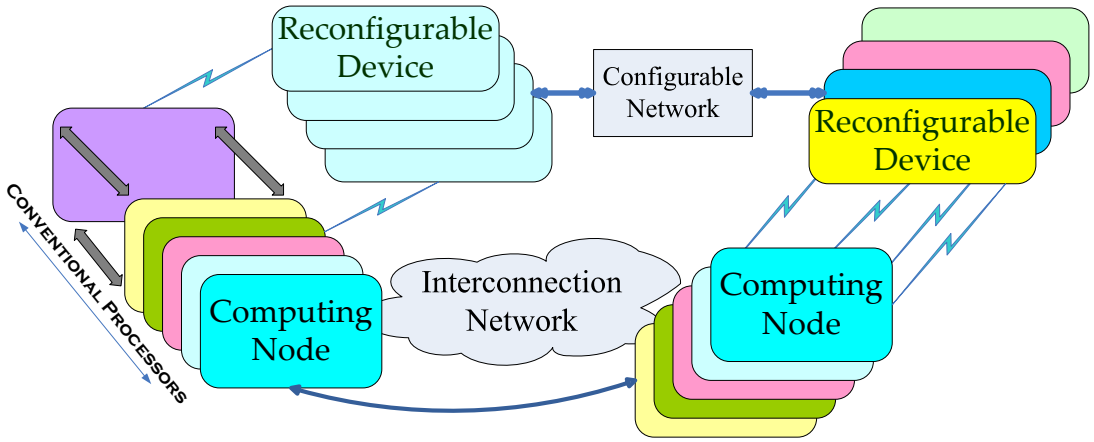

Fig. 2. High-Performance Reconfigurable Architecture

rately. The hardware side is expressed using Carte C or Carte Fortran as a separate function, compiled separately and linked to the compiled C (or Fortran) software side to form one application. Other hardware vendors use a third-party software tool, such as Impulse C, Handel-C, Mitrion C, or DSPlogic's RC Toolbox. However, these tools handle only the FPGA side of the application, and each machine has its own application interface to call those functions. At present, Mitrion C and Handel-C support the SGI RASC, while Mitrion C, Impulse C, and RC Toolbox support the Cray XD1. Currently, only a library-based parallel tool such as the MPI [61] can handle scaling an application beyond one node in a parallel system [80].

In a recent attempt to develop an HPRC system, Patel et al. [81] have proposed an architecture for a scalable computing machine built entirely using FPGA computing nodes. The machine enables designers to implement large-scale computing applications using a heterogeneous combination of hardware accelerators and embedded microprocessors spread across many FPGAs, all interconnected by a flexible communication network. Parallelism at multiple levels of granularity within an application can be exploited to obtain the maximum computational throughput. By providing a simple, abstracted communication interface with the objective of being able to scale to thousands of FPGA nodes, the proposed architecture appears to the programmer as a unified, extensible FPGA fabric. A programming model based on MPI is also presented to support partitioning an application into independent computing tasks that can be implemented on the architecture.

IV. Parallelism Limitations in Embedded Systems

The main problem with traditional parallel processing is that it requires the user to handle the parallelization, the synchronization, and the communication. In essence, the major problems of implementing multiprocessor applications are placed on the end user. Embedded reconfigurable architectures replicate many custom or semi-custom hardware components across the chip. It is often necessary to utilize different parallel processing techniques to take advantage of these multi-core resources.

Unfortunately, there are several problems that limit the multi-core embedded computing systems from reaching their potential. First, parallel processing is often hard to achieve. Writing parallel programs is extremely tedious and error prone. Automation tools for parallel programming (e.g., coarse grain parallelizing compilers and fine grain instruction level parallelism) have had limited success. A shared memory resource as used in many VLIW or symmetric multiprocessor-style architectures is popular, particularly in embedded DSP architectures. This is a similar concept to vector processing cores that are used in many processors, such as the Pentium (MMX) and PowerPC (AltiVec). Often the way applications are written can inadvertently hide parallelism from a compiler. In particular, the instruction level parallelism of even highly parallel codes is ultimately limited to what a compiler can find and this can be extremely low. These shared memory systems do not scale well for large number of processor cores, even if the ILP could be increased.

Second, many available sequential programs contain extremely limited parallelism (if any at all), making many of the cores sit idle, thus eliminating their benefits. Third, parallel execution can lead to tremendous overheads, such as coherence mechanisms and additional code required to manage the parallel execution.

Task Level Parallelism and *Workload Partitioning* have been and certainly continue to be two dominant software development issues for reconfigurable platforms, either for heterogeneous or homogeneous architectures. These issues are more critical on heterogeneous architectures, since specialized processors may

have additional constraints.

The most evolutionary stage in embedded application development is *Language Refinement*. A significant amount of work has been done on parallelizing functional languages such as C. However, the existing work can not be easily adapted to embedded reconfigurable computing as it may seems. It can partly be addressed to developer supervised parallelism exploitation. OpenMP [82] and MPI can be two evident examples.

Embedded Computing Languages need to offer a hybrid approach to identify task level parallelism, which were originally developed for general purpose computations. To put it in other words, we need to achieve high performance without the challenges of understanding the PEs architecture or sophisticated parallel programming techniques. There are some research platforms aimed to achieve this goal such as RapidMind [83], which intends to make programming processors as easy as single-threaded, single core programming, yet taking full advantage of all available resources. In summary, we can consider scalability of architecture, appropriate programming models and task control management to be the primary challenges for embedded system designs in the coming years.

V. Summary and Concluding Remarks

Embedded computing research is gaining popularity in recent years. Innovative concepts such as using embedded languages to exploit parallelism and tuning in performance, seems to be crucial to take an evolutionary step toward more convenient application development. Embedded reconfigurable systems have the potential to demonstrate the computing power for overcoming hard-to-solve problems. What makes the overall concept even more demanding is the fact that they can be steered in the direction of high performance computing as well as specific application accelerations. In this paper, we described the latest contributions in embedded reconfigurable computing, concerning parallelism in both hardware platforms and software facilities targeting these systems. As a summary, reconfigurable computing systems can lend themselves to high performance computing by taking advantage of parallelism at multiple levels of granularity, ranging from instruction through process level parallelism.

Considering parallelism utilization inherent in current applications one should note that some types of parallelism can be found statically, by just examining the program. Other opportunities can be found only dynamically, by executing the program. Static parallelism is easier to implement but does not cover all important sources of parallelism (maximum parallelism that can be achieved). Dynamic discovery of parallelism for a particular application can also be performed at many levels of abstraction ranging from instruction to task level and so on.

The authors believe that a major benefit of using a parallel programming paradigm/language, tailored to embedded reconfigurable computing, would be the ease of identifying and extracting opportunities for parallelism in an application which could result in maximal performance gain.

References

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[2] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[3] N. Weaver, V. Paxson, and J. M. Gonzalez, "The shunt: an fpga-based accelerator for network intrusion prevention," in *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2007, pp. 199–206.

[4] K. Jarvinen, M. Tommiska, and J. Skytta, "Comparative survey of high-performance cryptographic algorithm implementations on fpgas," *IEE Proceedings - Information Security*, vol. 152, no. 1, pp. 3–12, 2005.

[5] E. J. Swankoski, N. Vijaykrishnan, R. Brooks, M. Kandemir, and M. Irwin, "Symmetric encryption in reconfigurable and custom hardware," *International Journal of Embedded Systems*, vol. 1, no. 3/4, pp. 205–217, 2005.

[6] M.-H. Jing, Z.-H. Chen, J.-H. Chen, and Y.-H. Chen, "Reconfigurable system for high-speed and diversified aes using fpga," *Microprocess. Microsyst.*, vol. 31, no. 2, pp. 94–102, 2007.

[7] P. S. B. do Nascimento, M. E. de Lima, S. M. da Silva, and J. L. Seixas, "Mapping of image processing systems to fpga computer based on temporal partitioning and design space exploration," in *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*. New York, NY, USA: ACM Press, 2006, pp. 50–55.

[8] P. Graham and B. E. Nelson, "Reconfigurable processors for high-performance, embedded digital signal processing," in *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 1999, pp. 1–10.

[9] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *J. VLSI Signal Process. Syst.*, vol. 28, no. 1-2, pp. 7–27, 2001.

[10] A. Shoa and S. Shirani, "Run-time reconfigurable systems for digital signal processing applications: A survey," *J. VLSI Signal Process. Syst.*, vol. 39, no. 3, pp. 213–235, 2005.

[11] M. J. Myjak and J. G. Delgado-Frias, "A two-level reconfigurable architecture for digital signal processing," *Microelectron. Eng.*, vol. 84, no. 2, pp. 244–252, 2007.

[12] A. P. Kumar, V. Kamakoti, and S. Das, "System-on-programmable-chip implementation for on-line face recognition," *Pattern Recogn. Lett.*, vol. 28, no. 3, pp. 342–349, 2007.

[13] J. M. Cardoso and M. P. Véstias, "Architectures and compilers to support reconfigurable computing," *Crossroads*, vol. 5, no. 3, pp. 15–22, 1999.

[14] Z. Li and S. Hauck, "Configuration compression for virtex fpgas," in *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 147–159.

[15] M. Martina, G. Masera, A. Molino, F. Vacca, L. Sterpone, and M. Violante, "A new approach to compress the configuration information of programmable devices," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 48–51.

[16] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and

defragmentation," in *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM Press, 2002, pp. 187–195.

[17] I. Robertson and J. Irvine, "A design flow for partially reconfigurable hardware," *Trans. on Embedded Computing Sys.*, vol. 3, no. 2, pp. 257–283, 2004.

[18] E. Carvalho, N. Calazans, E. B. ao, and F. Moraes, "Padreh: a framework for the design and implementation of dynamically and partially reconfigurable systems," in *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*. New York, NY, USA: ACM Press, 2004, pp. 10–15.

[19] K. N. Vikram and V. Vasudevan, "Mapping data-parallel tasks onto partially reconfigurable hybrid processor architectures," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 9, pp. 1010–1023, 2006.

[20] M. Hübner and J. Becker, "Exploiting dynamic and partial reconfiguration for fpgas: toolflow, architecture and system integration," in *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*. New York, NY, USA: ACM Press, 2006, pp. 1–4.

[21] S. Hauck and A. Agarwal, "Software technologies for reconfigurable systems," Dept. of ECE, Northwestern Univ., Tech. Rep., 1996.

[22] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard, "Programmable active memories: reconfigurable systems come of age," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 4, no. 1, pp. 56–69, 1996.

[23] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, and H. A. E. Spaanenburg, "Seeking solutions in configurable computing," *Computer*, vol. 30, no. 12, pp. 38–43, 1997.

[24] A. DeHon and J. Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," in *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*. New York, NY, USA: ACM Press, 1999, pp. 610–615.

[25] M. Gokhale and P. S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 2005.

[26] B. E. Nelson, "The mythical ccm: In search of usable (and resuable) fpga-based general computing machines," in *ASAP '06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 5–14.

[27] H. AMANO, "A Survey on Dynamically Reconfigurable Processors," *IEICE Trans Commun*, vol. E89-B, no. 12, pp. 3179–3187, 2006.

[28] A. K. Jones, R. Hoare, D. Kusic, G. Mehta, J. Fazekas, and J. Foster, "Reducing power while increasing performance with supercisc," *Trans. on Embedded Computing Sys.*, vol. 5, no. 3, pp. 658–686, 2006.

[29] A. Gangwar, M. Balakrishnan, and A. Kumar, "Impact of intercluster communication mechanisms on ilp in clustered vliw architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 1, p. 1, 2007.

[30] H. Fatemi, B. Mesman, H. Corporaal, T. Basten, and R. Kleihorst, "Rc-simd: Reconfigurable communication simd architecture for image processing applications," *Journal of Embedded Computing*, vol. 2, no. 2, pp. 167–179, 2006.

[31] H. Fatemi, B. Mesman, H. Corporaal, T. Basten, and P. Jonker, "Run-time reconfiguration of communication in simd architectures." in *IPDPS*. IEEE, 2006.

[32] O. Muller, A. Baghdadi, and M. Jézéquel, "Asip-based multiprocessor soc design for simple and double binary turbo decoding," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 1330–1335.

[33] J. Robert L. Bocchino and V. S. Adve, "Vector llva: a virtual vector instruction set for media processing," in *VEE '06: Proceedings of the second international conference on Virtual execution environments*. New York, NY, USA: ACM Press, 2006, pp. 46–56.

[34] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for reconfigurable computing," in *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*. London, UK: Springer-Verlag, 1998, pp. 248–257.

[35] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha, "A spatial path scheduling algorithm for edge architectures," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 129–140, 2006.

[36] J.-E. Lee, K. Choi, and N. D. Dutt, "Instruction set synthesis with efficient instruction encoding for configurable processors," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 1, p. 8, 2007.

[37] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung, "Synthesis of application specific instructions for embedded dsp software," *IEEE Trans. Comput.*, vol. 48, no. 6, pp. 603–614, 1999.

[38] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*. New York, NY, USA: ACM Press, 2001, pp. 61–66.

[39] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini, "Shapes:: a tiled scalable software hardware architecture platform for embedded systems," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM Press, 2006, pp. 167–172.

[40] "SHAPES Project Website," http://shapes.atmelroma.it/twiki/bin/view/ShapesPublic.

[41] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Instruction scheduling for a tiled dataflow architecture," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 2006, pp. 141–150.

[42] C. R. Hardnett, K. V. Palem, and Y. Chobe, "Compiler optimization of embedded applications for an adaptive soc architecture," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. New York, NY, USA: ACM Press, 2006, pp. 312–322.

[43] M. S. Schlansker and B. R. Rau, "Epic: Explicitly parallel instruction computing," *Computer*, vol. 33, no. 2, pp. 37–45, 2000.

[44] R. Santos, R. Azevedo, and G. Araujo, "2d-vliw: An architecture based on the geometry of computation," in *ASAP '06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 87–94.

[45] A. Sudarsanam and S. Malik, "Simultaneous reference allocation in code generation for dual data memory bank asips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 2, pp. 242–264, 2000.

[46] R. Leupers and D. Kotte, "Variable partitioning for dual memory bank dsps," *icassp*, vol. 2, pp. 1121–1124, 2001.

[47] J. Cho, Y. Paek, and D. Whalley, "Fast memory bank assignment for fixed-point digital signal processors," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 1, pp. 52–74, 2004.

[48] J.-M. Daveau, T. Thery, T. Lepley, and M. Santana, "A retargetable register allocation framework for embedded processors," *SIGPLAN Not.*, vol. 39, no. 7, pp. 202–210, 2004.

[49] X. Zhuang, T. Zhang, and S. Pande, "Hardware-managed register allocation for embedded processors," in *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA: ACM Press, 2004, pp. 192–201.

[50] Y.-H. Lee and C. Chen, "An effective and efficient code generation algorithm for uniform loops on non-orthogonal

dsp architecture," *J. Syst. Softw.*, vol. 80, no. 3, pp. 410–428, 2007.

[51] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and compiling applications for rapid," in *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 1998, p. 116.

[52] M. Weinhardt and W. Luk, "Pipeline vectorization for reconfigurable systems," in *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 1999, p. 52.

[53] Q. Wang and D. M. Lewis, "Automated field-programmable compute accelerator design using partial evaluation," in *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 145–154.

[54] M. Budiu and S. C. Goldstein, "Fast compilation for pipelined reconfigurable fabrics," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 1999, pp. 195–205.

[55] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkarmark, X. Tian, and H. Saito, "Challenges in exploitation of loop parallelism in embedded applications," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM Press, 2006, pp. 173–180.

[56] H. Ziegler and M. Hall, "Evaluating heuristics in automatically mapping multi-loop applications to fpgas," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM Press, 2005, pp. 184–195.

[57] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. New York, NY, USA: ACM Press, 2006, pp. 136–146.

[58] Y. Dou, J. Xu, and G. Wu, "The Implementation of a Coarse-Grained Reconfigurable Architecture with Loop Self-pipelining," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4419, pp. 155–166, 2007.

[59] Z. Ma, F. Catthoor, and J. Vounckx, "Hierarchical task scheduler for interleaving subtasks on heterogeneous multiprocessor platforms," in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*. New York, NY, USA: ACM Press, 2005, pp. 952–955.

[60] Y. Krasteva, E. de la Torre, and T. Riesgo, "Reconfigurable Heterogeneous Communications and Core Reallocation for Dynamic HW Task Management," *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 873–876, 2007.

[61] "Message Passing Interface (MPI) Forum Home Page," http://www.mpi-forum.org.

[62] C. Shin, S.-W. Lee, and J.-L. Gaudiot, "Adaptive dynamic thread scheduling for simultaneous multithreaded architectures with a detector thread," *J. Parallel Distrib. Comput.*, vol. 66, no. 10, pp. 1304–1321, 2006.

[63] D. Scarpazza, P. Raghavan, D. Novo, F. Catthoor, and D. Verkest, "Software Simultaneous Multi-Threading, a Technique to Exploit Task-Level Parallelism to Improve Instruction-and Data-Level Parallelism," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4148, pp. 12–23, 2006.

[64] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, 2003.

[65] K. Bousias, N. Hasasneh, and C. Jesshope, "Instruction Level Parallelism through Microthreading-A Scalable Approach to Chip Multiprocessors," *The Computer Journal*, vol. 49, no. 2, pp. 211–233, 2006.

[66] S. M. Carta, D. Pani, and L. Raffo, "Reconfigurable coprocessor for multimedia application domain," *J. VLSI Signal Process. Syst.*, vol. 44, no. 1-2, pp. 135–152, 2006.

[67] R. Sasanka, M.-L. Li, S. V. Adve, Y.-K. Chen, and E. Debes, "Alp: Efficient support for all levels of parallelism for complex media applications," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 1, p. 3, 2007.

[68] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun, "A practical fpga-based framework for novel cmp research," in *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2007, pp. 116–125.

[69] C. Batten, R. Krashinsky, and K. Asanovic, "Scale control processor test-chip," Massachusetts Institute of Technology, Tech. Rep. MIT-CSAIL-TR-2007-003, January 2007.

[70] S. Wang and L. Wang, "Thread-associative memory for multicore and multithreaded computing," in *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*. New York, NY, USA: ACM Press, 2006, pp. 139–142.

[71] K. Wu, A. Kanstein, J. Madsen, and M. Berekovic, "MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4419, pp. 26–38, 2007.

[72] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews, "Enabling a uniform programming model across the software/hardware boundary," in *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 89–98.

[73] M. A. S. D. Fuentes, "A parallel programming model for a multi-fpga multiprocessor machine," Master's thesis, University of Toronto, 2006.

[74] T. El-Ghazawi, D. Buell, V. Kindratenko, and K. Gaj, "M03—reconfigurable supercomputing," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 217.

[75] T. El-Ghazawi, D. Bennett, D. Poznanovic, A. Cantle, K. Underwood, R. Pennington, D. Buell, A. George, and V. Kindratenko, "Reconfigurable supercomputing—is high-performance reconfigurable computing the next supercomputing paradigm?" in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 71.

[76] "SRC computers, inc. - programmer-friendly reconfigurable computing systems," http://www.srccomputers.com/.

[77] "SGI RASC technology," http://www.sgi.com/products/rasc/.

[78] "CRAY inc. website," http://www.cray.com/.

[79] "Linux networx - the linux supercomputing company, howpublished = http://www.linuxnetworx.com/,."

[80] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko, "Guest editors' introduction: High-performance reconfigurable computing," *Computer*, vol. 40, no. 3, pp. 23–27, 2007.

[81] A. Patel, C. A. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow, "A scalable fpga-based multiprocessor," *FCCM*, vol. 0, pp. 111–120, 2006.

[82] "OpenMP Official Website," http://www.openmp.org.

[83] "The RapidMind Platform," http://www.rapidmind.net/.