MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture

Kehuai Wu, Jan Madsen
Dept. of Informatics
and Mathematic Modelling
Technical University of Denmark
{kw, jan}@imm.dtu.dk

Andreas Kanstein
Freescale Semiconductor
a.kanstein@freescale.com

Mladen Berekovic
IMEC, Belgium
berekovi@imec.be

Abstract

The coarse-grained reconfigurable architecture ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) and its compiler offer high instruction-level parallelism (ILP) to applications by means of a sparsely interconnected array of functional units and register files. As high-ILP architectures achieve only low parallelism when executing partially sequential code segments, which is also known as Amdahl's law, this paper proposes to extend ADRES 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. This article presents details of the enhanced architecture and results obtained from an MPEG-2 decoder implementation that exploits a mix of thread-level parallelism and instruction-level parallelism.

1 Introduction

The ADRES architecture template is a datapath-coupled coarse-grained reconfigurable matrix[3]. As shown in figure 1, it resembles a very-long-instruction-word (VLIW) architecture coupled with a 2-D coarse-grained heterogeneous reconfigurable array (CGRA), which is extended from the VLIW's datapath. As a template, ADRES can have various number of VLIW function unit and various size of CGRA. Applications running on the ADRES architecture are partitioned into control-intensive code and computation-intensive kernels by the DRESC compiler [2]. The control-intensive fraction of the application is executed on the VLIW, while the computation-intensive parts, the loops or kernels are modulo-scheduled on the CGRA. By seamlessly switching the architecture between the VLIW mode and the CGRA mode at run-time, statically partitioned and scheduled applications can be run on the ADRES with a high number of instructions-per-clock (IPC).

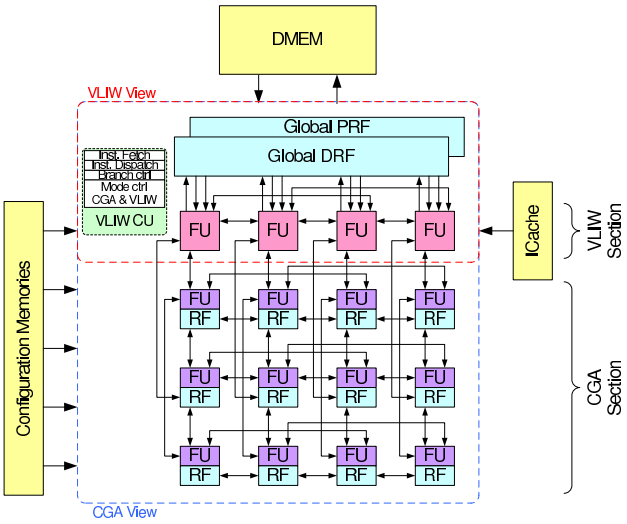

Figure 1. ADRES architecture template and its two operation modes

Based on the observation of our previous work [1], most MPEG2 decoder kernels can be scheduled on the CGRA with the IPC ranging from 8 to 43. We have also observed that some modulo-scheduled kernels' IPC do not scale very well when the size of the CGRA increases. Some of our most aggressive architectures have the potential to execute 64 instructions per clock cycle, but few kernels can utilize such a large CGRA efficiently: Either the inherent ILP is low and cannot be increased efficiently even with loop unrolling, or the code is too complex to be scheduled efficiently on so many units due to resource constraints, for example the number of memory ports. Also, the CGRA is mostly idle when executing sequential code in VLIW mode. The more sequential code is executed, the lower

the achieved application's average IPC. In conclusion, even though the ADRES architecture is highly scalable, we are facing the challenge of getting more parallelism out of many applications.

Since the ADRES architecture has a large amount of functional units (FU), the most appealing approach for increasing the application parallelism would be simultaneous multi-threading (SMT)[4]. However, due to the static nature of the DRESC tool chain and the complexity of the CGRA, such dynamic/speculative[6][5] threading is inappropriate for the ADRES architecture. Our threading approach identifies an application's coarse-grained parallelism based on the static analysis results. If properly reorganized and transformed at programming time, multiple kernels with weak dependency can be efficiently parallelized by the application designer.

Our long-term goal is to apply static explicit simultaneous multithreading on the ADRES architecture. The current single-threaded architecture and its matching compilation tools has been demonstrated on an FPGA prototype. Starting from the single-threaded architecture and compilation tool, we carried out a demonstrative dual-threading experiment on the MPEG2 benchmark. Through this experiment, we have proven that the multithreading is feasible for the ADRES architecture.

The rest of the paper is organized as follows. Section II discusses the threading solution we will achieve in the long run. Section III describes the proof-of-concept experiment we carried out, and what design issues must be addressed in terms of architecture and compilation methodology. Section IV pin-points the design issues in the future and how we can address them. Section V concludes our work.

2 ADRES multithreading

2.1 Overview

We propose a scalable partitioning-based threading approach for ADRES. The rich resource on the ADRES architecture allows us to partition a large ADRES into two or more sub-arrays, each of which can be viewed as a down-scaled ADRES architecture and be partitioned further down the hierarchy, as shown in figure 2. As long as the partitioning process doesn't violate the resource constraint on each partition, the VLIW and CGRA can be split further into sub-arrays, each of which executes a programmer-defined thread.

Each thread has its own resource requirement. A thread that is easy to parallelize requires more computation resources, thus executing it on a larger partition results in the optimal use of the ADRES array and vise versa. A globally optimal application design demands that the programmer knows the IPC of each part of the application, so that he can find an efficient array partition for each thread.

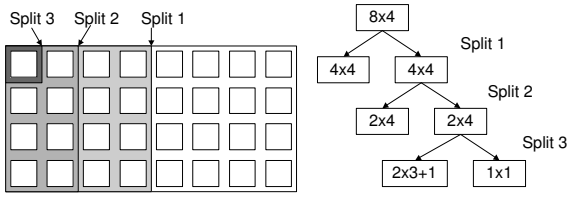

Figure 2. Scalable partitioning-based threading

The easiest way to find out how many resources is required by each part of a certain application is the profiling. Programmer starts from a single-threaded application and profiles it on a large single-threaded ADRES. From the profiling results, kernels with low IPC is identified as the high-priority candidates for threading. Depending on the resource demand of the threads, programmer statically plans on how and when the ADRES should split into partitions during application execution. When the threads are well-organized, the full ADRES array can be optimally utilized.

2.2 Architecture design issue

The FU array on the ADRES is heterogeneous. There exists dedicated memory units, Floating point units, special arithmetic units and control/branch units on the array that constrain the partitioning. When partitioning the array, we have to guarantee that the program being executed on certain partition can be scheduled. This requires that any instruction invoked in a thread to be supported by at least one of the functional unit in the array partition. The well-formed partitions usually have at least one VLIW FU that can perform branch operation, one FU that can perform memory operation, several FP/arithmetic units if needed, and several FUs that can handle general operations.

On the ADRES architecture, the VLIW register file (RF) is a critical resource that can not be partitioned easily. The recent ADRES architecture employs the clustered register files that has previously been adapted to the VLIW architectures[7] [8]. If we prohibit the RF bank to be shared among several threads, RF cluster can be partitioned with the VLIW/CGRA, and the thread compilation can be greatly simplified. In case a single RF is used, the register allocation scheme must be revised to support the constrained register allocation, as suggested in our proof-of-concept experiments.

The ADRES architecture with multiple memory units requires ultra-wide memory bandwidth. Multi-bank memory has been adapted to our recent architecture[10], and proven

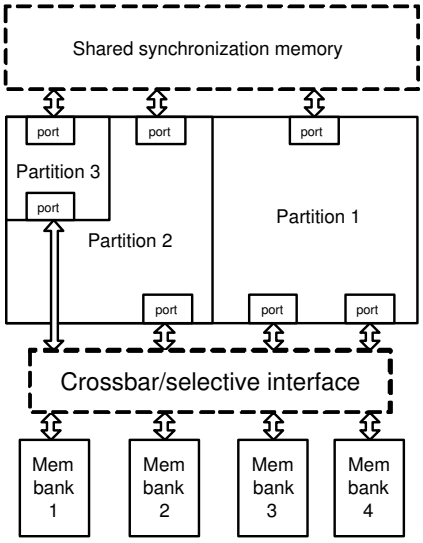

Figure 3. Synchronization memory

to cope nicely with our static data-allocation scheme. There are currently two ways to organize the multi-bank memory hierarchy of ADRES template. The first organization employs a crossbar between the memory and the algorithm core. This approach offers a scratchpad style of memory presentation to all the load/store units, but the cost of implementing the crossbar is obviously an issue. The second organization is the selective interfacing between load/store units and a subset of memory banks. Under this organization, each load/store unit can only access a part of the memory space, and the static data allocation is the key to guarantee the correctness of the program execution. Depending on the requirement of the application, we choose a suitable organization to interface the memory to the load/store units.

The memory-based synchronization is the most common synchronization technique for systems with cache-coherent memory hierarchy. While the scratchpad memory organization is less of a problem for ADRES, the selective interfacing is non-coherent, thus data memory organized in this manner cannot be used for building synchronization mechanism. We propose to add a small shared memory that can be accessed by at least one memory unit each partition purely for synchronization purpose, as shown in figure 3. Since the memory doesn't have to be large to implement the basic synchronization primitives, it can be a small register file with modest amount of ports.

Dedicated synchronization primitives like register-based semaphore or pipe can also be adapted to ADRES template. These primitives can be connected between pairs of functional units that belongs to different thread partitions. Synchronization instruction can be added to certain functional units as intrinsics, which is supported by the current DRESC tools.

In the single-threading ADRES architecture, the program counter and the dynamic reconfiguration counter is controlled by a finite-state-machine (FSM) type control unit. When implementing the multithreading ADRES, we need an extendable control mechanism to match our hierarchically partitioned array. As shown in figure 4, we duplicate the FSM type controller and organized the controllers in a hierarchical manner. In this multithreading controller, each possible partition is still controlled by an FSM controller, but the control path is extended with two units called merger and bypasser. The merger and bypasser form a hierarchical master-slave control that is easy to manage during program execution.

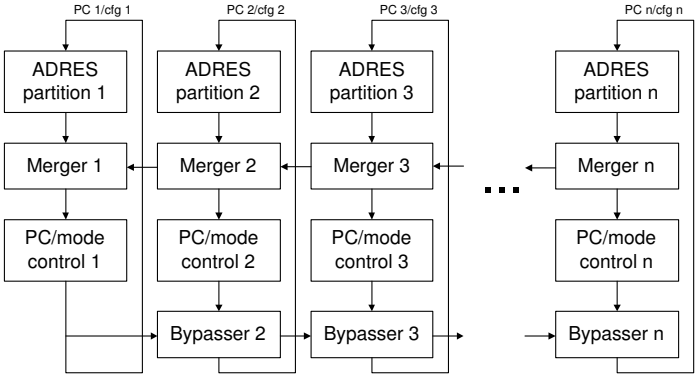

Figure 4. Hierarchical multithreading controller

The principle of having such a control mechanism is as following. Suppose we have an ADRES architecture that can be split into two halves for dual threading, while each half has it's own controller. In order to reuse the controller as much as possible, we prefer each of the controller to control a partition of the ADRES when the program is running in dual threaded mode, but one of the controller can also take full control of the whole ADRES when the program is running in the single-threaded mode. By assigning one of the controller to control the whole ADRES, we created the master. When the ADRES is running in the single-thread mode, the master controller also sense the signal from the slave partition and merge them with the master partition's signal for creating global control signal. At the same time, the slave partition should bypass any signal generated from the local controller and following the global control signal generated from the master partition. When the ADRES is running in the dual-threaded mode, the master and slave controller completely ignores the control signals coming from the other partition and only responds to the local signals. This strategy can be easily extended to cope with further partitioning.

2.3 Multithreading methodology

The multithreading compilation tool chain is extended from our previous DRESC compiler[2]. With several modifications and rearrangements, most parts of the original DRESC tool chain, e.g. the IMPACT frontend, DRESC modulo scheduler, assembler and linker can be reused in our multithreading tool chain. The most significant modifications in our tool chain are made on the application programming, the architecture description and the simulator.

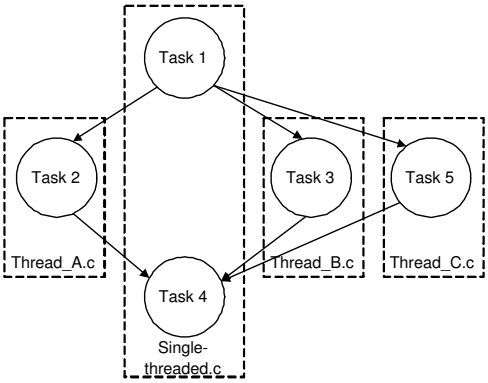

Figure 5. Source code reorganization

Before the threaded application can be compiled, the application needs to be reorganized. As shown in figure 5, the application is split into several c-files, each of which describes a thread that is executed on a specific partition, assuming the application is programmed in c. Such reorganization takes modest effort, but makes it easier for the programmer to experiment on different thread/partition combinations to find the optimal resource budget.

The multithreading ADRES architecture description is extended with the partition descriptions, as shown in figure 6. Similar to the area-constrained placement and routing on the commercial FPGA, when a thread is scheduled on an ADRES partition, the instruction placement and routing is constrained by the partition description. The generated assembly code of each thread goes though the assembling separately, and gets linked in the final compilation step.

The simulator reads the architecture description and generates an architecture simulation model before the application simulation starts. As shown in figure 4, each partition has its own controller, thus the generation of the controller's simulation model depends on the partition description as well. Furthermore, the control signal distribution is also partition-dependent, thus requires the partition description to be consulted during simulation model generation.

Some other minor practical issues needs to be addressed in our multithreading methodology. The most costly problem is that different partitions of the ADRES are conceptually different ADRES instances, thus a function compiled for a specific partition cannot be executed on any other partitions. When a function is called by more than one thread, multiple partition-specific binaries of this function has to be stored in the instruction memory for different caller. Secondly, multiple stacks need to be allocated in the data memory. Each time the ADRES splits into smaller partitions due to the threading, a new stack need to be created to store the temporary data. Currently, the best solution to decide where the new stack should be created is based on the profiling, and the thread stacks are allocated at compile time. And finally, each time the new thread is created, a new set of special purpose registers needs to be initialized. Several clock cycles are needed to properly initial the stack points, the return register, etc. immediately after the thread start running.

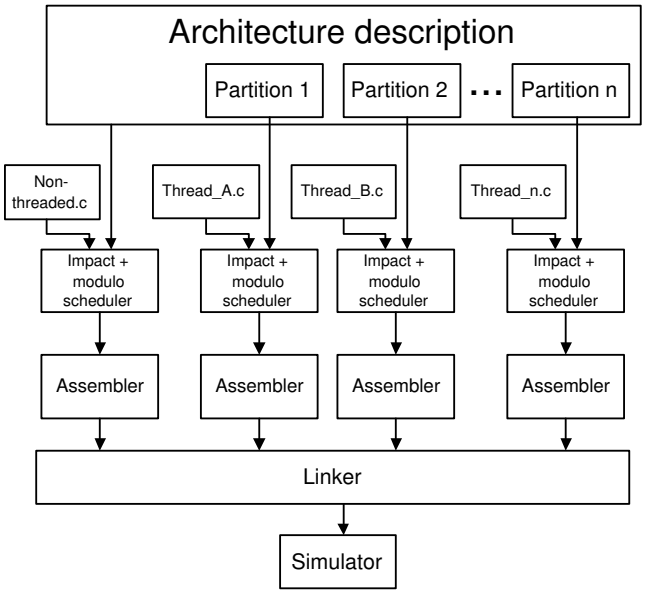

Figure 6. Multithreading compilation tool chain

3 Proof-of-concept experiment

3.1 Application and methodology

Our proof-of-concept experiment achieves dual-threading on the MPEG2 decoder. The MPEG2 decoder can be parallelized on several granularities[9], thus is a suitable application to experiment on. We choose the Inverse Discrete Cosine Transform (IDCT) and Motion Compensation (MC) as two parallel threads, and reorganized the MPEG2 decoder as shown in figure 7. The

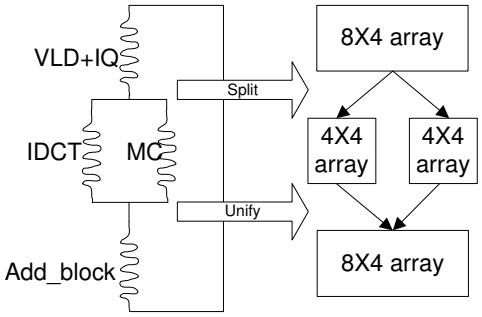

Figure 7. Threading scenario on MPEG2 decoder

decoder starts its execution on an 8x4 ADRES, executes the Variable Length Decoding (VLD) and Inverse Quantization (IQ), and switches to the threading mode. When the thread execution starts, the 8x4 ADRES splits into two 4x4 ADRES arrays and continues on executing the threads. When both threads are finished, the two 4x4 arrays unify and continue on executing the add block function. We reorganized the MPEG2 program as described in figure 5, and added "split" and "unify" instructions as intrinsics. These instructions currently do nothing by themselves, and is only used to mark where the thread mode should change in the MPEG2's binary code. These marks are used by the split-control unit at run time for enabling/disabling the thread mode program execution.

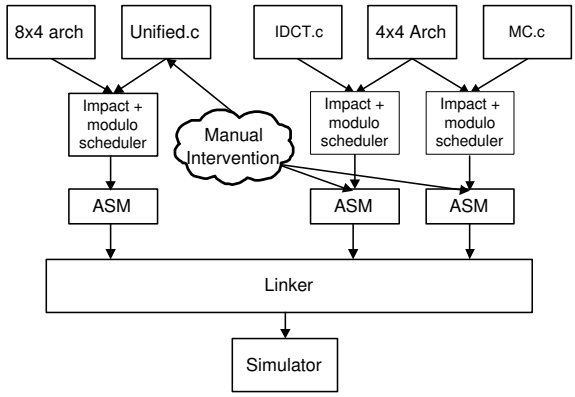

Figure 8. Experimental Dual-threading compilation flow

The current dual-threading compilation flow is shown in figure 8. The lack of partition-based scheduling forces us to use two architectures as the input to the scheduling. The 8x4 architecture is carefully designed so that the left and the right halves are exactly the same. This architecture is the execution platform of the whole MPEG2 binary. We also need a 4x4 architecture, which is a helping architecture that is compatible to either half of the 8x4 array. This architecture is used as a half-array partition description of the 8x4 architecture. With these two architectures in place, we compile the single-threaded c-file and the threads on the 8x4 architecture and the 4x4 architecture, respectively. The later linking stitches the binaries from different parts of the program seamlessly.

3.2 Memory and register file design

The memory partitioning of the threaded MPEG2 is shown in figure 9. The instruction fetching (IF), data fetching (DF) and the configuration-word fetching (CW) has been duplicated for dual-threading. The fetching unit pairs are step-locked during single-threaded program execution. When the architecture goes into the dual-threading mode, the fetching unit pairs split up into two sets, each of which is controlled by the controller in a thread partition.

During the linking, the instruction memory and data memory are divided into partitions. Both the instruction and configuration memory are divided into three partitions. These three partition pairs store the instructions and configurations of single-threaded binaries, IDCT binaries and MC binaries, as shown on figure 9. The data memory is divided into four partitions. The largest data memory partition is the shared global static data memory. Both single-threaded and dual-threaded program store their data into the same memory partition. The rest of the data memory is divided into three stacks. The IDCT thread's stack grows directly above the single-threaded program's stack, since they uses the same physical controller and stack pointer. The base stack address of the MC thread is offset to a free memory location at linking time. When the program execution goes into dual-threading mode, the MC stack pointer is properly initialized at the cost of several clock cycles.

In the future, we aim at partitioning the clustered register file among the array partitions so that each thread has it's own register file(s). However, due to the lack of a partitioning-based register allocation algorithm at the current stage, the partitioning approach is not very feasible. We experiment on the ADRES architecture with single global register file and go for the duplication based approach to temporary accommodate the register file issue. As shown in figure 10, a shadow register file has been added into the architecture. When the single-threaded program is being executed, the shadow register file is step-locked with the primary register file. When the program initiate the dual-thread execution, the MC thread gets access to the shadow register file and continues the execution on the array partition and shadow register file. When the program resume to

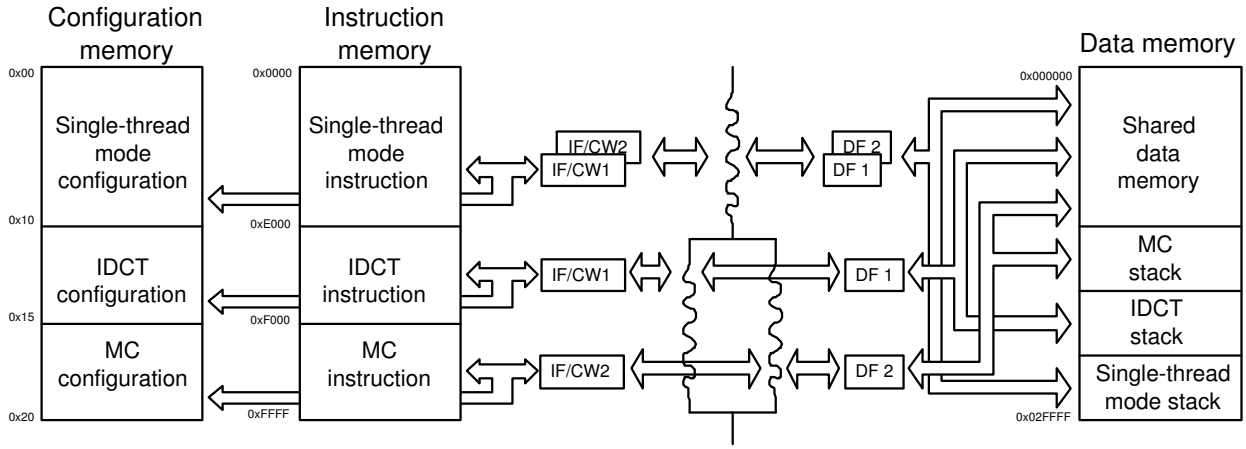

Figure 9. Dual-threading memory management

the single threaded execution, the shadow register file become hidden again. The MPEG2 program is slightly modified so that all the data being shared between threads and all the live-in and live-out variables are passed through the global data memory.

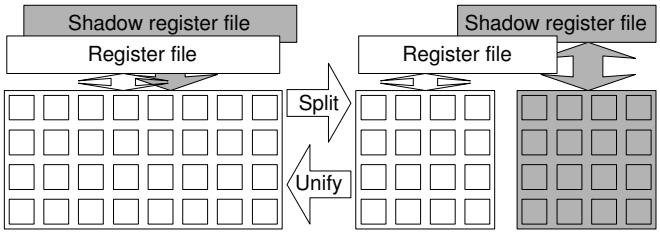

Figure 10. Shadow Register file setup

3.3 Control mechanism design

The scalable control concept in figure 4 has been verified in our simulation model. By having our control scheme tested on the dual-threading, we are positive that this scheme can be extended to a certain scale, and the control unit simulation model generation can be automated.

During the program linking, we identify where the "split" and "unify" instructions are stored in the instruction memory. These instructions' physical addresses mark the beginning and the ending point of the dual-threading mode. During the simulation model generation, these instructions' addresses are stored in a set of special-purpose registers in a split-control unit. After the program starts executing, the program counter's (PC) values are checked by the the split-control unit in each clock cycle. When the program counter reach the split point, the split-control unit sends control signals to the merger and bypasser to enable the threading mode. After the program goes into the threaded mode, the split-controller waits for both threads to join in by reaching the PC value where the "unify" instructions are stored. The first thread that joins in will be halt till the other thread finish. When the second thread eventually joins in, the split-control unit switch the ADRES array back to single-threaded mode, and the architecture resumes to the 8x4 array mode.

When an application gets more complicated and has multiple splitting/unifying point, the current approach will become more difficult to manage, thus the future architecture will only rely on the instruction decoding to detect the "split" and "unify" instructions. The split-control unit will be removed in the future, and part of its function will be moved into each partition's local controller.

3.4 Simulation result

The simulation result shows that the threaded MPEG2 produces the correct image frame at a slightly faster rate. Table 1 shows the clock count of the first 5 image frame decoded on the same 8x4 ADRES instance with and without threading. The dual-threaded MPEG2 is about 10% faster than the single-thread MPEG2, even if we are not aiming for any performance gain at the current stage.

As we have observed, the performance gain is mostly achieved from the ease of modulo-scheduling on the smaller architecture. When an application is scheduled on a highly complicated CGRA, many redundant instructions are added into the kernel for routing purpose. Now the kernels are scheduled on a smaller CGRA, both the CGRA utilization and the VLIW mode IPC has been improved.

frame number	single-thread cc count	dual-threaded cc count
1	1874009	1802277
2	2453279	2293927
3	3113150	2874078
4	3702269	3374421
5	4278995	3861978

Table 1. clock cycle count of single and dual threaded MPEG2 on the same architecture

4 Future work

The current tool chain has been experimented on the dual-threading. From our experience, we are certain that the tools satisfy the minimum requirements on the applications that do not exchange too many data between threads during run-time. Even if some limits currently exists, e.g. incapable of creating threads with parameters, some modification on the application source code can accommodate most problems with very low performance penalties.

However, the current tool chain requires significant amount of nonrecurrent work. Each time the architecture or the application changes, almost everything needs to be redone, starting from reassessing the application to generating the simulator. Also, once the simulation produces erroneous results, the debugging could be extremely time consuming. Furthermore, manual work will be time-consuming if more threads are to be added onto the ADRES.

The most critical update on the current tool chain is the partition based scheduling. Having this ready gives the programmer the chances to quickly experiment on the various possible way of partitioning an ADRES instance, and tries to explore the combination of threads and partition. Also, the architecture partitioning will be more scalable, flexible and error prune. The original ADRES scheduling is conceptually an instruction-set-constrained instruction-level placement and routing algorithm. Extending such an algorithm into area-constrained placement and routing requires reasonable amount of work.

The resource-constrained register allocation algorithm, multiple stack and multiple special-purpose register supports are another group of urgently needed update. Having these functions ready in the new linker enables us to shift to the cluster register file based architecture, and ease the programmer's effort on reorganizing the application.

The shared synchronization memory described in figure 3 is also a critical design issue. The number of ports that the synchronization memory can have is the scaling bottleneck of our approach. We would like to avoid having a large number of ports on the synchronization memory, since the worst-case memory delay is often dependent on how many ports a cell needs to drive. We expect that each partition should at least have one or two memory fetch unit that can access the synchronization memory, thus it is questionable to have more than 4 partitions on the array. Dedicated synchronization primitives like pipe and semaphore can help to ease this problem at a cost of extra architecture design time.

Some other minor future design issues include automating the simulation model generation; expanding the ADRES instruction set for "split" and "unify" instruction support; redesigning the split-control unit, etc. But we haven't found out any pitfalls that can hinder our progress to realize the automated methodology in the near future.

5 Conclusions

Our MPEG2 experiment has given us ample knowledge on the Multithreading ADRES architecture. The simulation results shows that the MPEG2 has gain 10% of speed up, only from the ease of scheduling on smaller architecture. We expect that more speed gain can be achieved with more elaborated future benchmarking. We believe that our threading approach is adequate for ADRES architecture, is practically feasible, and can be scaled to certain extend. Based on the success of our proof-of-concept experiments, we have very positive view on the future of MT-ADRES.

References

[1] B. Mei *A Coarse-grained Reconfigurable Architecture Template and its Compilation Techniques*, Ph.D. thesis, IMEC, Belgium.

[2] B. Mei, S. Vernalde, D. Verkest, H. D. Man and R. Lauwereins *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*, International Conference on Field Programmable Technology, Hong Kong, December 2002, pages 166-173.

[3] B. Mei, S. Vernalde, D. Verkest, H. D. Man and R. Lauwereins *DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures*, FPL 2003

[4] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, Dean M. Tullsen *SIMULTANEOUS MULTITHREADING: A Platform for Next-Generation Processors*, Micro, IEEE Volume 17, Issue 5, Sept.-Oct. 1997 Page(s):12 - 19

[5] E. Ozer, T.M. Conte *High-performance and low-cost dual-thread VLIW processor using Weld architecture paradigm* , IEEE Transactions on Parallel and Distributed Systems, Volume 16, Issue 12, Dec. 2005 Page(s):1132 - 1142

[6] H. Akkary, M.A. Driscoll *A dynamic multithreading processor* ,31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998. MICRO-31. Proceedings. 30 Nov.-2 Dec. 1998 Page(s):226 - 236

[7] J. Zalamea, J. Llosa, E. Ayguade, M. Valero *Hierarchical clustered register file organization for VLIW processors*, International Parallel and Distributed Processing Symposium, 2003. Proceedings. 22-26 April 2003 Page(s):10 pp.

[8] A. Capitanio, N. Dutt, and A. Nicolau. *Partitioned register files for VLIWs: A preliminary analysis of tradeoffs.*, The 25th Annual International Symposium on Microarchitecture, pages 103–114, Dec. 1992.

[9] E. Iwata, K Olukotun *Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm*, Stanford University Computer Systems Lab Technical Report CSL-TR-98-771, September 1998.

[10] B. Mei , S. Kim, R. Pasko, *A new multi-bank memory organization to reduce bank conflicts in coarse-grained reconfigurable architectures*, IMEC, Technical report, 2006