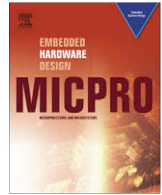
Microprocessors and Microsystems 33 (2009) 290–294

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro



# Mapping of nomadic multimedia applications on the ADRES reconfigurable array processor

Mladen Berekovic [a,*], Andreas Kanstein [b], Bingfeng Mei [a], Bjorn De Sutter [a]

[a] IMEC, B-301 Leuven, Belgium
[b] Freescale Semiconductor, 31023 Toulouse Cedex, France

ARTICLE INFO

*Article history:*
Available online 20 February 2009

*Keywords:*
Coarse-grain reconfigurable arrays
ADRES
DRESC
Processor architecture
DSP
Multimedia
MPEG
H.264
Reconfigurable computing

ABSTRACT

This paper introduces the mapping of MPEG video decoders on ADRES, IMEC's new coarse-grain reconfigurable and fully C-programmable array processor that targets nomadic devices. ADRES is a flexible template that allows the instantiation of many different processor versions. An XML-based architecture description language allows a designer to easily generate different processor instances with full compiler support by specifying different values for the communication topology, the number and size of local register files and functional units and supported instruction set. ADRES supports a VLIW-like programming model with a pure VLIW mode for legacy code, and a (coarse-grain reconfigurable) array mode with very high parallelism for the processing of compute intensive loops. We demonstrate the mapping of two video decoders MPEG-2 and AVC, and discuss the performance trade-offs for two critical kernels: IDCT and integer transform. As a result, an ADRES based system can perform AVC decoding in CIF resolution with less then 50 MHz on a $4 \times 4$ array processor.

© 2009 Elsevier B.V. All rights reserved.

## 1. Introduction

A new class of programmable processor architectures is emerging for demanding DSP applications such as video coding: coarse-grained reconfigurable architectures (CGRAs). While many CGRAs were proposed in recent years [4] none of them have yet been widely adopted, partially due to the difficult programming models, and partially due to the vast overuse of resources when compared to other DSP processors. Another typical problem is the difficult interfacing between the array and the host processor, where the control flow part of the application code is running. These issues are addressed by a novel CGRA called architecture for dynamically reconfigurable embedded systems (ADRES) and by its compiler technology called dynamically reconfigurable embedded system compiler (DRESC) [9]. Firstly, the ADRES architecture tightly couples a very-long instruction word (VLIW) processor and a coarse-grained array by providing two functional views on the same physical resources. The VLIW part offers an easy path for the mapping of complex applications, that is absent in other published CGRA implementations. Furthermore, the array part offers unprecedented loop accelerations. Secondly the DRESC compiler framework assures that applications written in C can be easily mapped onto VLIW and array mode. The sharing of a central registerfile between these two modes, that also serves as a storage for live-in and live-out variables for the loop mode, minimizes communication and mode-switching costs and enables the compiler to seamlessly generate code for both modes, including the data transfer operations. Finally, ADRES is a template instead of a concrete processor architecture. With the retargetable compilation support from DRESC, architectural exploration becomes possible to discover better architectures or design domain-specific architectures.

We mapped two key video applications, namely MPEG-2 [5,14] and H.264 [6,13] decoding, on ADRES [11]. Firstly, the applications have been compiled for the VLIW-view. Next, the IDCT, which is also used in MPEG-4 [7] and integer transform kernels are accelerated on the array part of ADRES. The results for these are discussed in detail, and compared to benchmarks for a state-of-the art VLIW–DSP processor, TI's TMS320C64× [1].

This paper is organised as follows. In Section 2 we first present the architecture of the ADRES reconfigurable array processor in Section 2 and the corresponding DRESC compiler in Section 3. Then, in Section 4, we illustrate our application mapping methodology that is specific for this type of reconfigurable array and apply it for MPEG in Section 5. The mapping results are presented in Section 6 and compared to other stat-of-the-art processors in Section 7. Section 8 presents hardware implementation results and finally, in Section 9 our conclusions are discussed.

## 2. The ADRES CGRA

The ADRES architecture template, as shown in Fig. 1, consists of an array of basic components, including FUs, register files (RFs) and

* Corresponding author.
*E-mail address:* berekovic@ida.ing.tu-bs.de (M. Berekovic).

0141-9331/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2009.02.008

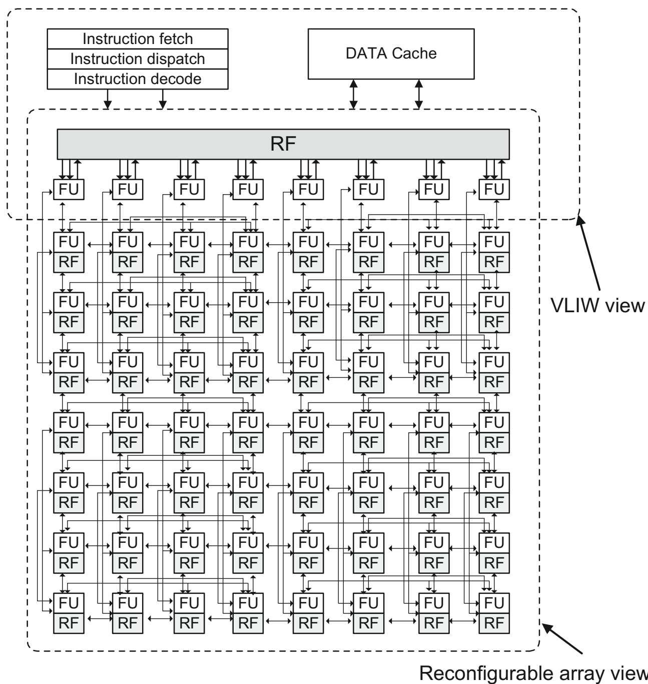



**Fig. 1.** Architecture of the ADRES coarse-grain reconfigurable array.

routing resources. The top row can act as a tightly coupled VLIW processor in the same physical entity. The two parts of ADRES share the same central register file and load/store units. The computation-intensive kernels, typically dataflow loops, are mapped onto the reconfigurable array by the compiler using the modulo scheduling technique to implement software pipelining and to exploit the highest possible parallelism, whereas the remaining code is mapped onto the VLIW processor. The data communication between the VLIW processor and the reconfigurable array is performed through the shared RF and shared memory. The array mode is controlled from the VLIW controller as an infinite loop between two (configuration memory) address pointers with a data dependent loop exit signal from within the array that is handled by the compiler.

The array contains three types of basic components: functional units (FUs), storages resources such as register files and memory blocks, and routing resources that include wires, muxes and busses. The ADRES architecture is a flexible template that can be freely specified by an XML-based architecture specification language as an arbitrary combination of those elements.

Fig. 2 shows a detailed datapath of an ADRES FU. In contrast to FPGAs, the FU in ADRES performs coarse-grained operations on 32 bits of data, e.g., ADD, MUL, shift. To remove the control flow inside loops, the FU supports predicated operations for conditional execution. A good timing is guaranteed by buffering the outputs in a register for each FU. The results of the FU can be written to a local register file (RF), which is usually small and has less ports than the shared RF, or routed directly to the inputs of other FUs. The multiplexors are used for routing data from different sources. The configuration RAM acts as a (VLIW-) instruction memory to control these components. It stores a number of configuration contexts locally, which are loaded on a cycle-by-cycle basis. Fig. 2 shows only one possible data path, as different heterogeneous FUs are quite possible.

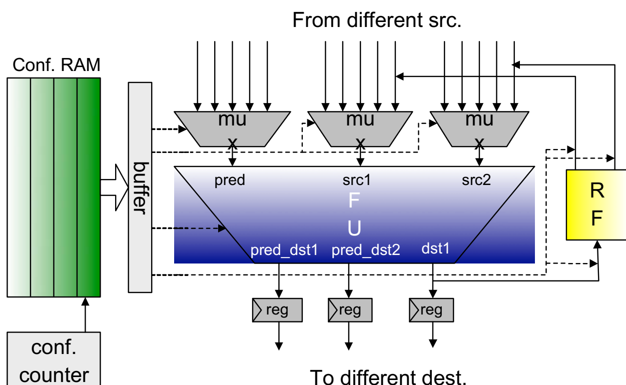



**Fig. 2.** ADRES array node (FU).

## 3. DRESC compiler framework

For a complex architecture like ADRES, an automatic design methodology and tools are essential. Therefore we developed the ADRES architecture together with its own compiler framework, called dynamically reconfigurable embedded system compiler (DRESC). The compiler framework is depicted in Fig. 3. A design starts from a C-language description of the application. The profiling/partitioning step identifies the candidate compute intensive loops (kernels) for mapping on the reconfigurable array based on execution time and possible speed-up. Source-level transformations are used to make the kernel software pipelineable (see the following section) and to maximize the performance. In the next step, we use IMPACT, a VLIW compiler framework [3,15], to parse the C code and do some analysis and optimization. IMPACT emits an intermediate representation, called Lcode, which is used as input for scheduling. As can be seen on the right-hand side of Fig. 3, the target architecture is described in an XML-based language. This high level parameterized description of the architecture allows a designer to quickly specify different architecture variation. The parser and abstraction steps transform the architecture to an internal, more detailed, graph representation. Taking program and architecture representation as input, a novel modulo scheduling algorithm is applied to achieve high parallelism for the kernels, whereas traditional ILP scheduling techniques are applied to discover the available moderate parallelism for the non-kernel code. The communication between these two parts is automatically identified and handled by our tools. Finally, the tools generate scheduled code for both reconfigurable array and VLIW. Three kinds of simulator are generated from the architecture description and the scheduled code and are used to obtain quality metrics for the architecture instance under test.

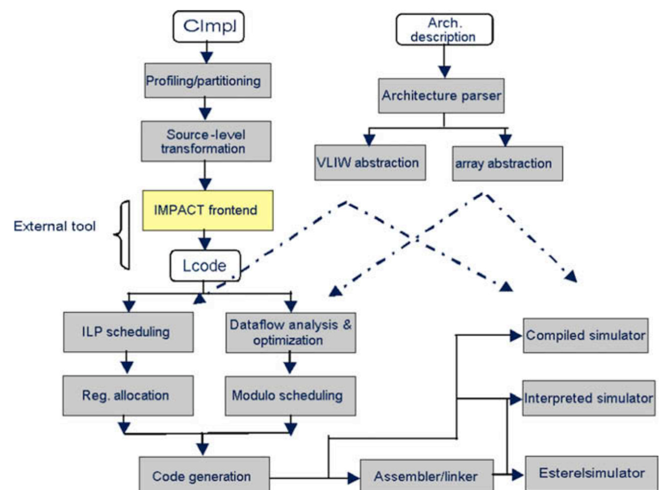



**Fig. 3.** DRESC compiler framework.

The core technology of the compiler is the modulo scheduling algorithm that is capable of mapping loops onto the ADRES architecture in a highly parallel way [10]. Modulo scheduling is a widely used software pipelining technique [8]. The objective of modulo scheduling executes multiple iterations of the same loop in parallel. This is achieved by constructing a schedule for one iteration of the loop such that this same schedule is repeated at regular intervals with respect to intra- and inter-iteration dependences and resource constraints. This interval is termed initiation interval (II), and it specifies after how many cycles the execution of a new iteration of the loop is initiated. Because of this, the initiation interval is inversely proportional to the performance and it is used to indicate the performance of the scheduled loop. Applied to coarse-grained architectures, complexity of modulo scheduling increases drastically as it needs to combine three sub-problems, placement, routing and scheduling, in a modulo constrained space. To illustrate the problem, a simple data dependence graph (DDG) is shown, representing a loop body, and a $2 \times 2$ array (Fig. 4a). The scheduled loop is depicted in Fig. 4b, which is a space–time representation of the scheduling space. The $2 \times 2$ array is flattened to $1 \times 4$ for convenience of drawing. The dashed lines represent a routing possibility between the FUs. Placement determines on which FU of a 2D (two-dimensional) array to place an operation. Scheduling determines in which cycle to execute that operation. Routing connects the placed and scheduled operations according to their data dependences. The schedule on the $2 \times 2$ array is shown in Fig. 4c, where initiation interval is equal to one. FU1, FU2, FU3 and FU4 are configured to execute n2, n4, n1 and n3, respectively. By overlapping different iterations of a loop, an instruction per cycle (IPC) of four is achieved in this simple example. As a comparison, it takes three cycles to execute one iteration in a non-pipelined schedule due to the data dependences, corresponding to an IPC of 1.33, no matter how many FUs are in the array.

Internally, our modulo scheduling algorithm utilizes a graph-based architecture representation, called modulo routing resource graph (MRRG), to model resources in a unified way, expose routing possibilities and enforce modulo constraints. The algorithm is based on congestion negotiation and simulated annealing methods. Starting from an invalid schedule that overuses resources, it tries to reduce overuse over time until a valid schedule is found. One main advantage of the DRESC framework is its flexibility. The tools are designed to be retargetable within the ADRES template. An architecture instance can be easily described in the XML-based description language. Without any changes, the compiler and simulator are automatically retargetable to different instances.

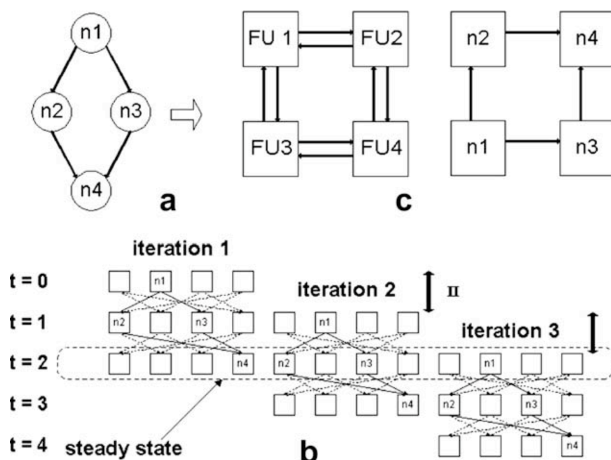



**Fig. 4.** (a) Data dependence graph that will be mapped to $2 \times 2$ reconfigurable array; (b) schedule with initiation interval (II) = 1; and (c) resulting configuration DRESC compiler framework.

## 4. Application mapping methodology

The input for mapping an application to ADRES is C code, possibly enhanced with intrinsics. There are some source-level transformations the programmer must perform to achieve an efficient mapping of loops to the CGRA.

The first step is to determine the critical kernels of an application, which is done through profiling. Then, the loops within the kernels have to be made compilable with DRESC: the loop body may not contain function calls and branches, and it must have a single exit. Fortunately IMPACT already supports this when the code is compiled to hyperblocks: the code blocks will have a single exit, and predicates are being used to replace code branches. The programmer sometimes has to support this by simplifying the control flow and manually inlining functions in the loop body.

Then the loop should be compiled and run on the compiled-code simulator for a first evaluation. The compiler will report on the schedule length, the initiation interval (II) or number of cycles used for executing the loop body, and the resource usage. The simulator will provide statistics on the actual performance of the mapped loop. The data will indicate whether the iteration count is high enough and whether there are resource constraints which should be removed. The combination of a low iteration count and a long schedule indicates overhead, because a proportionally higher amount of cycles is being spent in the loop prologue and epilogue.

There are several techniques that can be used for increasing the performance. Loop coalescing combines a nested loop with its enclosing one to generate a larger, longer loop. Another aspect to consider is whether two loops can be combined, like the horizontal and the vertical passes of a 2D filter. The increase of the loop body might come with an advantage as more independent instructions become available for achieving a better pipelining. Loop unrolling has a similar effect, which also helps when the resources in the array are under-used. A special case is the trade-off between memory accesses and additional computations, because an array typically has far less memory interface than computation resources. Here the programmer can trade-off additional computations against memory accesses.

In some cases, a loop has already been optimized for a different architecture, and loop unrolling, the insertion of intrinsics or the usage of look-up tables have to be undone. It might further be beneficial to split a loop into two, to reduce the number of memory accesses. Then, with deeper application knowledge, the processing flow can be optimized to generate even larger loop iterations.

Another dimension of the mapping is the architecture template. If the specific ADRES implementation is also to be defined, the programmers, together with the system architects, can consider adding or removing register files, connections, and even special instructions. The latter are supported in the tool flow in the form of intrinsics that are programmed by means of function calls. Besides lowering the instruction count, the benefits of using intrinsics are in reducing the schedule length by combining dependent instructions, and in making more efficient use of the data path by exploiting sub-word parallelism.

## 5. Mapping multimedia applications

Two key benchmarks from video compression have been mapped on ADRES: MPEG-2 and H.264 decoding. Both codes have been compiled for ADRES with several key kernels running on the

**Table 1**
Performance results of MPEG-2 and H.264 decoder on an $8 \times 8$ ADRES with intrinsics.

| | MPEG-2 decoding (CIF) | H.264 decoding (CIF) |
|---|---|---|
| ADRES $8 \times 8$ with intrinsics | 27 MHz | 56 MHz |

array. Table 1 shows the overall performance of these applications on an ADRES template with $8 \times 8$ (total of 64) FUs.

Two most demanding kernels from a commercial H.264 basic profile decoder, the deblocking and the combined interpolation and motion compensation have been optimized for executing them on the array part of ADRES. The motion compensation had been implemented in 16 kernels for luma and four for chroma. To optimize the loops, first the clipping, which had been done with a look-up table, has been replaced with a min/max computation, and then loop coalescing has been applied. Then several loops were merged for code-size reasons. Finally, intrinsics replaced the clipping and the shift with rounding operations, and the loops were unrolled for better pipelining. In a second approach, the loops were split for vertical and horizontal interpolation, for reducing the number of memory accesses and the code-size. Undoing loop unrolling and removing table look-up code were also the first steps on optimizing the deblocking filter. Then the task was to define loops with decent numbers of iterations, but with a reasonably sized loop bodies. The problem with the deblocking filter is that just applying loop coalescing results in a loop body which contains too many code branches. The implementation then has overly much predicated code, as well as a very-long schedule, if it compiles at all.

## 6. Results and architecture exploration

For another two of the key kernels, IDCT from MPEG-2 and integer transform from H.264, several experiments have been carried out using different architecture templates, varying array sizes and instruction sets. Five different $8 \times 8$ architecture templates were used. First, we used a standard $8 \times 8$ template with the connectivity shown in Fig. 1: nearest neighbor, four-hop and busses. Secondly, an $8 \times 8$ array with routing adds a second independent output register to each FU, thus offering an independent path for the routing of data. Thirdly, the $8 \times 8$ array with p2p interconnect replaces the busses with individual point-to-point connections. Finally, the $4 \times 4$ and the $14 \times 8$ arrays are standard arrays with varied sizes. Furthermore, IDCT kernels with three different instruction sets were tested: without intrinsics (Idct_un_opti0.c), with simple intrinsics (e.g., saturated instructions: Idct_un_opti1.c) and with more complex intrinsics functions as found in the TMS320C64 processor core (Idct_un_opti3.c). Table 2 summarizes the characteristics of these different IDCT implementations.

Table 3 summarizes the performance results for the $8 \times 8$ IDCT kernel. What can be seen is that with intrinsics a speed-up of almost a factor of two can be achieved, while the smaller $4 \times 4$ array is only slower by a factor of two and is hence more area and power-efficient, especially when combined with intrinsics. Interestingly, the better routing offers the same speed-up as the larger array size ($14 \times 8$).

Table 4 shows the performance results for the $4 \times 4$ integer transform. The numbers are for the calculation of a single macroblock, i.e., for 16 iterations of the transform. Two kernels were

**Table 2**
Characteristics of the IDCT and Itrans kernels.

| | Idct_unopt | Idct_opt11 | Idct_opt3 |
|---|---|---|---|
| Total number of instruction | 1544 | 1184 | 704 |
| Min cycle single loop (dependence) | 60 | 50 | 38 |

**Table 3**
Performance results of MPEG-2 and H.264 on ADRES.

| | idct_un_opti0.c | idct_un_opti1.c | idct_un_opti3.c |
|---|---|---|---|
| 8x8 default | 61 | 45 | 37 |
| 8x8 & rout | 47 | 38 | 30 |
| 8x8 & p2p | 45 | 44 | 30 |
| 14x8 | 45 | 37 | 30 |
| 4x4 | 144 | 118 | 72 |
| | Speedup = 2.4 | Speedup = 2.6 | Speedup |

**Table 4**
Performance results of the inverse integer transform in H.264 decoder in cycles (performed on 16 blocks) on different array architectures.

| | $4 \times 4$ Itrans, merged loops, per macroblock | $4 \times 4$ Itrans, separated loops per macroblock |
|---|---|---|
| $8 \times 8$ default | 141 | 308 |
| $8 \times 8$ and route | 123 | |
| $14 \times 8$ | 115 | |
| $4 \times 4$ | 263 | 312 |

used, one with merged horizontal and vertical loops, and one with separated loops that is performing significantly worse. No intrinsic instructions were used. What can be seen from the numbers is that this algorithm does not scale as well with additional resources as the IDCT, Hence an optimized $4 \times 4$ ADRES will execute this most efficiently, i.e., with highest utilization of the available FUs.

## 7. Comparison with commercial VLIW DSPs

Fig. 5 shows a comparison for the IDCT between an $8 \times 8$ ADRES core and other known benchmarks. The numbers for StarCore, TI and ADRES were obtained by compiling the code with the respective compilers and cycle-true simulation. In the case of TI, C and assembler code from TIs web page were used. The cycle count for PACT is taken from [2] and for Morphosys is taken from [12].

These architectures can be categorised into two different groups: very-long-instruction-word (VLIW) processors that execute compiler-translated C code and coarse-grain reconfigurable array processors that are typically programmed in assembler or on a lower level by setting muxes and switches. While TI falls into the first category, PACT and Morphosys belong to the second. StarCore is a kind of an exception since it employs a more traditional DSP architecture with less instruction-level parallel processing while ADRES actually combines these two groups by supporting a (VLIW-) compiler for a coarse-grained array processor.

What can be seen from these numbers is that ADRES beats TI's TMS320C64× cores and StarCore LLC' SC1400 by a large margin, but it even beats a hand-coded assembler implementation with heavy use of intrinsics on a TI TMS320C64× by a factor of four. It also outperforms the more sophisticated but hand-programmed array processor from PACT and Morphosys.

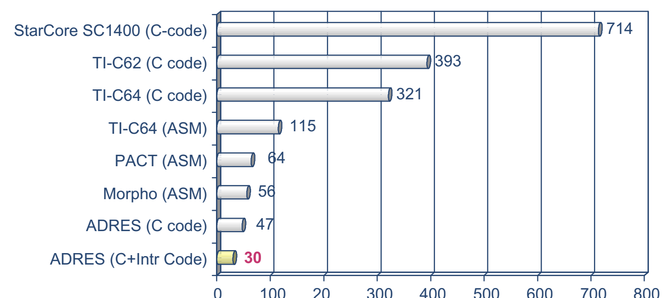



**Fig. 5.** Performance comparisons between ADRES ($8 \times 8$) and other DSP solutions on $8 \times 8$ IDCT.

**Table 5**
Implementation sizes of different ADRES configurations compared to TI's TMS320C64× (90 nm).

| | 4 × 4 FUs, 16 RFs, 16 MULs (mm$^2$) | 8 × 8 FUs, 64 RFs, 64 MULs (mm$^2$) | TI C64 (mm$^2$) |
|---|---|---|---|
| Processor core | 1 | 4 | 2 |
| Processor core + L1 | 3 | 6 | 4 |
| Processor core + L1 + L2 ctrl + peripheral | 9 | 12 | 10 |
| Processor core + L1 + L2 + peripheral | 19 | 22 | 20 |

## 8. Hardware implementation results

Table 5 shows the implementation sizes of different ADRES configurations compared to TI's TMS320C64× [1]. ADRES offers better performance and scalability for lower cost, together with a retargetable C compiler. While the TMS320C64× also comes with a C compiler, the difference in results obtained for assembly code and C code in Fig. 5 clearly show that that processor cannot be programmed in C efficiently.

Synthesis experiments show that ADRES will reach 600 MHz in 90 nm technology, which compares with 1 GHz reached by the TMS320C64×. Power-efficiency will be in the range of 30 MOPS/mW. The cache configurations use 16 Kb of instruction and data cache and 1 MB of L2 cache. Since ADRES is a templatized core for system-on-chip designs, the proper cache configuration for ADRES will be selected upon the demands of the target applications, such as AVC. This contrasts with the TI processor that incorporates large caches to enable its standalone operation for more general purpose signal processing applications.

## 9. Conclusions and future work

Coarse-grained reconfigurable architectures (CGRAs) have been emerging as potential processor architectures for future programmable DSP systems in recent years. The main hurdle for mainstream acceptance has been the lack of good compilers. The CGA template ADRES together with its retargetable C compiler framework, DRESC, are addressing this issue. The ADRES architecture tightly couples a VLIW processor and a reconfigurable array, resulting in improved performance, ease-of-programming, lower communication costs, resource sharing and as a result, better power efficiency. Furthermore the architecture template can be adapted to certain classes of applications.

Experiments in mapping MPEG-2 and H.264 decoding demonstrate that complex applications can be mapped with competitive performance on ADRES architecture templates. A detailed analysis for IDCT shows that ADRES can perform better than state-of-the art DSP processors. Future work will include more analysis on the power consumption versus performance aspects for different array architectures and instruction sets.

## Acknowledgements

This research has been performed in the context of IMECs M4 Research Program, which is partly funded by Samsung and Freescale Semiconductors.

## References

[1] S. Agarwala, T. Anderson, A. Hill, M.D. Ales, R. Damodaran, P. Wiley, S. Mullinnix, J. Leach, A. Lell, M. Gill, A. Rajagopal, A. Chachad, M. Agarwala, J. Apostol, M. Krishnan, Quang Duc Bui, Quang An, N.S. Nagaraj, T. Wolf, T.T. Elappuparackal, A 600 MHz VLIW–DSP, IEEE Journal of Solid-State Circuits 37 (11) (2002) 1532–1544.
[2] J. Becker, A. Thomas, Scalable processor instruction set extension, IEEE Design and Test of Computers 22 (2) (2005) 136–148.
[3] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter, W.-M. Hwu, IMPACT: an architectural framework for multiple-instruction-issue processors, in: Proceedings of the 18th International Symposium on Computer Architecture (ISCA), 1991, pp. 266–275.
[4] R. Hartenstein, A decade of reconfigurable computing: a visionary retrospective, in: Proceedings of Design, Automation and Test in Europe (DATE), 2001, pp. 642–649.
[5] ISO/IEC 13818-1/-2/-3, Information Technology—Generic Coding of Moving Pictures and Associated Audio: Systems/Video/Audio, 1996.
[6] ISO/IEC 14496-10, Coding of Audiovisual Objects – Part 10: Advanced Video Coding; also Known as ITU-T Recommendation for H.264 – Advanced Video Coding for Generic Audiovisual Services, 2003.
[7] ISO/IEC 14496-2 (1999)/Amd. 1 (2000), Coding of Audio–Visual Objects—Part 2: Visual; and Amendment 1: Visual Extensions.
[8] M.S. Lam, Software pipelining: an effective scheduling technique for VLIW machines, in: Proceedings of ACM Conference on Programming Language Design and Implementation, 1988, pp. 318–327.
[9] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins, ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix, in: Proceedings of Field-Programmable Logic and Applications, 2003, pp. 61–70.
[10] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins, Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling, in: Proceedings of Design, Automation and Test in Europe (DATE), 2003, pp. 296–301.
[11] B. Mei, S. Vernalde, D. Verkest, R. Lauwereins, Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: a case study, in: Proceedings of Design, Automation and Test in Europe (DATE), 2004, pp. 1224–1229.
[12] H. Singh, M.-H. Ming, G. Lu, F. Kurdahi, N. Bagherzadeh, E.M.C. Filho, Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications, IEEE Transactions on Computers 49 (5) (2000) 465–481.
[13] T. Wiegand, G.J. Sullivan, G. Bjontegaard, A. Luthra, Overview of the H.264/AVC video coding standard, IEEE Transactions on Circuits and Systems for Video Technology (CSVT) 13 (7) (2003) 560–576.
[14] Official MPEG Website, at <http://www.chiariglione.org/mpeg/>.
[15] IMPACT Compiler, <http://www.crhc.uiuc.edu/Impact/>.