Implementation of a Coarse-Grained Reconfigurable Media Processor for AVC Decoder

B. Mei (bennet@imec.be), A. Kanstein (a.kanstein@freescale.com), B. De Sutter (desutter@imec.be), T. Vander Aa (vanderaa@imec.be), S. Dupont (duponts@imec.be) and M. Wouters (wouters@imec.be)

Abstract. ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) is a templatized coarse-grained reconfigurable processor architecture. It targets at embedded applications which demand high-performance, low-power and high-level language programmability. Compared with typical VLIW-based DSP, ADRES can exploit higher parallelism by using more scalable hardware with support of novel compilation techniques. We developed a complete tool-chain, including compiler, simulator and HDL generator. This paper describes the design case of a media processor targeting at H.264 decoder and other video tasks based on the ADRES template. The whole processor design, hardware implementaiton and application mapping are done in a relative short period. Yet we obtain C-programmed real-time H.264/AVC CIF decoding at 50 MHz. The die size, clock speed and the power consumption are also very competitive compared with other processors.

1. Introduction

Nowadays consumers would like to have one portable device that combines the functionality of a mobile phone, personal digital assistant, digital camera,... This requires a processor that can combine the very different performance requirements of these functionalities with the energy efficiency that is needed for a battery powered device. At the same time the increasing cost of ASICs is driving the designers' choice towards more flexible solutions. Because of this, coarse-grained reconfigurable architectures (CGRAs) have become increasingly important in recent years. Various architectures have been proposed (Hartenstein, 2001) that are composed of an array of Functional Units (FUs). An FU is a hardware operator that is capable of executing word- or subword-level operations (e.g. addition, multiplication, shift) instead of the bit-level operations found in common FPGAs. This *coarse* granularity greatly reduces delay, area, power and configuration time compared with FPGAs, however, at the expense of flexibility (Hartenstein, 2001). On the other hand, compared with traditional embedded processors such as DSPs (digital signal processors), these architectures offer potential to achieve higher performance and power-efficiency because of their massive amount of resources and less expensive architectural features.

However, these architectures generally lack good tool support due to their complexity. They are programmed manually or have limited automatic tool

support. On the other hand, modern multimedia and wireless communication applications are becoming increasingly complex. It is extremely difficult for application developers to program such complex architectures manually. This is one of important reasons that CGRAs have yet to make it into mainstream applications.

In past, we have developed ADRES (Architecture for Dynamically Reconfigurable Embedded Systems (Mei et al., 2003a)), which features a unique concept of combining a VLIW mode and an reconfigurable array mode into the same architecture. The VLIW mode is used for exploit instruction-level parallelism, while the array mode is designed to accelerate loops in a pipelined way. A novel modulo scheduling algorithm was proposed to solve the key compilation problem of mapping loops onto partially connected array (Mei et al., 2003b). A prototype compiler called DRESC (Dynamically Reconfigurable Embedded Systems Compiler) is developed based on the algorithm.

In this paper, we present a more complete tool flow based on recent developments. It meets requirements from application development downto hardware verification: a compiler, assembler, linker and several simulators. It also includes tools to automatically generate synthesizable VHDL from high-level architecture description and help fast power simulation. Based on the toolset, we demonstrate how a high-performance and low-power media processor for video applications is designed as an instance of ADRES template. With our retargetable toolset support, it is very easy to exploit different design options to achieve our goal. We also show the hardware implementation results of the media processor. It achieves clock speed of 300MHz and die size of only $3.6mm^2$. More importantly, it meets our low-power goal for target applications. The whole media processor and the mapped H.264/AVC decoder have been verified and demonstrated on an FPGA-based platform. The design of the media processor proves our architecture and toolset are mature enough to handle complex application such as an H.264/AVC decoder.

The remainder of this paper is organized as follows. Section 2 gives description of the ADRES architecture template and its complete toolset. Section 3 discusses design choices that were made to obtain the required operation frequency and to obtain high enough IPC (instruction per cycle). Experimental results are reported in Section 4. Section 5 discusses some related work. Finally, conclusions are drawn and future work is discussed in Section 6.

2. ADRES Reconfigurable Processor Template and Toolset

This section describes the ADRES processor template and its toolset.

2.1. ARCHITECTURE DESCRIPTION

The ADRES processor template is shown in Figure 1. It consists of an array of basic components, including FUs, register files (RFs) and routing resources. At the highest abstraction level, it tightly couples a VLIW processor and a reconfigurable array in the same physical entity. The identified computation-intensive kernels, typically loops, are mapped onto the reconfigurable array, 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 access. ADRES is a flexible template specified by an XML-based architecture specification language, which is integrated into the DRESC compiler framework.

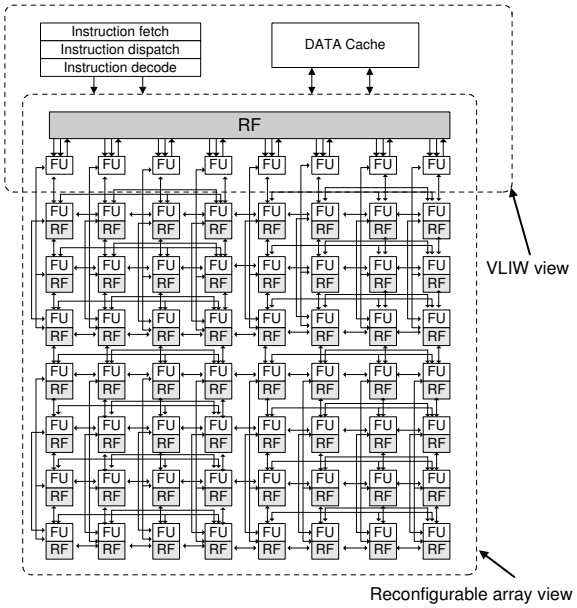


Figure 1. ADRES architecture template

Inside the ADRES processor (Figure 1), we find many basic components, including computational resources, storage resources and routing resources. The computational resources are functional units (FUs), which are capable of executing a set of operations. The storage resources mainly refer to the register file (RFs) and memory blocks, which can store intermediate data. The routing resources include wires, multiplexors and busses. Basically, computational resources and storage resources are connected by the routing resources in the ADRES array. This is similar to other CGRAs. The ADRES array is a flexible template instead of a concrete instance. Figure 1 only shows one instance of the ADRES array with a topology resembling the MorphoSys ar-

chitecture (Singh et al., 2000). An XML-based description language specifies the chosen ADRES instances (see Section 2.4.1).

Figure 2 shows the details of an example datapath. The FU performs *coarse-grained* operations, i.e., operations on 8, 16, 32 or 64-bit words. The FU supports predicated operations to remove control flow. This is needed since the scheduling algorithm used does not support loops with internal control flow (see Section 2.3.1). To guarantee timing, the outputs of FUs are required to be buffered by an output register. The results of the FU can be written to the local RF, which is usually small and has less ports than the shared RF, or routed to other FUs. The multiplexors are used for routing data from different sources. The configuration RAM provides bits to control these components. It stores a number of configuration contexts locally, which can be loaded on a cycle-by-cycle basis. The configurations can also be loaded from the memory hierarchy at the cost of extra delay if the capacity of the local configuration RAM is not sufficient. Figure 2 shows only one possibility of how the datapath can be constructed. Very different instances are possible. For example, the output ports of a RF can be connected to input ports of several neighboring FUs. The ADRES template has much freedom to build an instance out of these basic components.

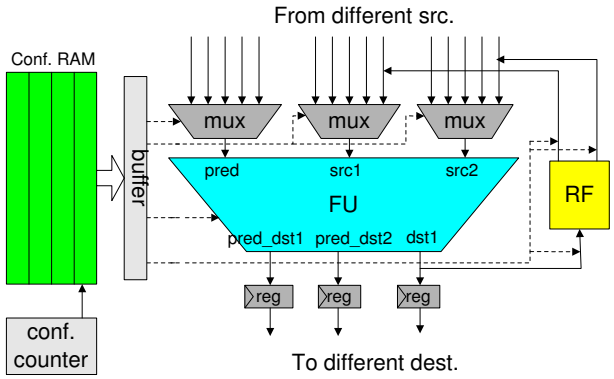


Figure 2. An example of detailed datapath

2.2. Execution and Configuration Model

The most important feature of the ADRES architecture is the tight coupling between a VLIW processor and a coarse-grained reconfigurable array. Since VLIW processors and CGRAs use similar components like FUs and RFs, a natural thought is to make them share those components. The whole ADRES architecture has two virtual functional views: a VLIW processor and a reconfigurable array. These two virtual views share some physical resources because their executions will never overlap with each other thanks to the processor/co-processor execution model. This execution model also greatly

simplify compiler development though at expenses of losing potential performance gain using concurrent execution model.

For the VLIW processor, several FUs are allocated and connected together through one multi-port register file. The FUs used in VLIW are generally more powerful. For example, some of them have to support the branch and subroutine call operations. The instructions of the VLIW processor are loaded from the main instruction memory hierarchy. This requires typical steps like instruction fetching, dispatching and decoding. For the reconfigurable array part, all the resources, including the RF and FUs of the VLIW processor, form a big 2D array. The array is connected by partial routing resources. Dataflow-like kernels are mapped to the array in a pipelined way to exploit high parallelism. The FUs and RFs of the array are simpler than those of the VLIW processor. The communication between these two virtual views is through the shared VLIW register file and memory access. The sharing is in the time dimension so that it does not increase the hardware cost. For example, it does not require more ports in the VLIW RF.

In the VLIW mode, the configuration is performed as in all other VLIW processors: in each cycle, an instruction is fetched from the instruction memory hierarchy and executed. In the array mode, the configuration contexts are fetched from the on-chip configuration memory. Each kernel may use one or more consecutive contexts. If the configuration memory is big enough to accommodate all the kernels, all these kernels only need to be loaded once at start-time. Afterward, the reconfiguration can be done on cycle-by-cycle basis. If the configuration memory is not big enough for all the kernels, one or more existing kernel has to be carefully chosen and discarded in order to load a new kernel from the main instruction-memory hierarchy. This is known as the kernel scheduling problem (Maestre et al., 2001). Proper algorithms can minimize the reconfiguration overhead (Maestre et al., 2001). However, this aspect is not addressed in this paper.

2.3. BASIC COMPONENTS

2.3.1. *Functional Units*

An FU can perform a set of operations. In ADRES, only fixed-point operations are supported because they are considered sufficient for typical telecommunication and multimedia applications. All FUs are fully pipelined so that one instruction can be issued at each cycle even when the latency of that instruction is more than one cycle. Different implementations may lead to different latency, which can be specified in the architecture description (Section 2.4.1) and is supported by the compiler.

The supported instruction set is very close to that of a standard RISC processor. Typical instructions supported include arithmetic operations (ADD, SUB, MUL), logic operations (AND, OR, etc.), and compare operations (CMP,

PRED). The VLIW FUs, typically the first row of FUs, also support load/store and control operations (LOAD, STORE, Branch, JMP). The DRESC toolset allows a user to define and use intrinsic functions such as *max* and *min*, which are often used in signal processors. The intrinsic functions can be used just as normal functions, and will be compiled to special instructions supported by certain FUs.

Predicated execution is introduced in the FUs in order to remove control-flow using *if-conversion* (Allen et al., 1983) and do other transformations. Each FU has three source operands: *pred*, *src1* and *src2*. *pred* is a 1-bit signal (Figure 2). If it is 1, the operation is executed; otherwise, the operation is nullified. *src1* and *src2* are normal data source operands. To enhance the routability of the ADRES array, the FU is augmented with swapping logic for *src1* and *src2* operands. Therefore, the operands of all operations can be switched, even though some operations, such as shifts, are not commutative. This feature increases the scheduling freedom. The FU also has three destination operands: *pred_dst1*, *pred_dst2* and *dst*. *pred_dst1* and *pred_dst2* are complementary 1-bit predicates holding the results of special comparison operations. *dst* is the normal output operand.

Furthermore, the FUs are enhanced with routing operations that allow src operands to be copied to the destination ports of FUs in array mode. The SEL1 and SEL2 instructions copy data from *src1* or *src2* to *dst*, similar to MOV operation. Besides the main operation to be executed on an FU, so-called extra operations can be executed at the same time to route predicate data. The extra operations CON1, CON2 and CON12 take care of the predicate part by copying a predicate from *pred* to *pred_dst1*, *pred_dst2* or both.

2.3.2. *Register Files*

Register files (RFs) are used in the ADRES architecture as one of the main storage resources. We support different types of register files: various amount of read and write ports are supported, as well as 1-bit predicate and 32-bit data register files, rotating/non-rotating/mixed register files.

Normally the shared VLIW register file requires multiple ports. For each FU of VLIW, at least 2 read ports and 1 write ports are required. Thus, for the 3-issue VLIW implemented on the FPGA platform, a 9-port RF is used (see also Section 3). For distributed RFs inside the array, typically only simple RFs are required with one read port and one write port. Simpler RFs are not only faster and smaller, but also consume significantly less power.

Rotating register files (RRFs) are available to handle special requirements of software pipelining (Rau et al., 1992). RRFs basically implement hardware-based register renaming. Each physical RF address is calculated by adding a virtual RF address and a value from the iteration counter of executed loops. Hence, the different instances of the same variable in different iterations are assigned to different physical registers, thus avoiding name clashes and the

need for software-based renaming. In ADRES, a RF can be specified as rotating, non-rotating or mixed. A mixed RF contains a section of rotating registers and a section of non-rotating registers. In the ADRES XML template, an architecture designer can use these RFs freely with the only constraint that at least one non-rotating RF or one non-rotating RF section should be available in the VLIW execution mode.

2.3.3. *Routing Network*

The routing networks consist of a data network and a predicate network. The data network routes the normal data among FUs and RFs, while the predicate network directs 1-bit predicate signals. These two networks do not necessarily have the same topology. Because of their different data widths, they cannot overlap.

The basic routing resources are mux, point-to-point wire, and bus. Additionally, an FU can be used for routing both data and predicate using routing operations discussed above. Arbitrary latency can be specified for these routing resources, in which case latches are inserted. A hardware designer can exploit this freedom to the brake up critical paths in his design, thus achieving the target clock speed (See section 3).

2.4. DRESC TOOLSET

For a complex architecture like ADRES, extensive tool support is essential for designers to be really able to use it. Therefore we have developed the ADRES architecture together with a complete set of tools, centered around a compiler called DRESC (Dynamically Reconfigurable Embedded System Compiler (Mei et al., 2002)). The compiler framework is depicted in Figure 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 make the kernel software pipelineable (see next paragraph) and to maximize the performance. In the next step, we use IMPACT, a VLIW compiler framework (IMPACT,), to parse the C code and to perform some analyses and optimizations. IMPACT emits an intermediate representation, called *Lcode*, which is used as input for scheduling. The XML-based architecture description that was described before is used as an input for most of the lower level steps in the flow. The parser and abstraction steps transform the architecture to an internal, more detailed, graph representation (Mei et al., 2003b). 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

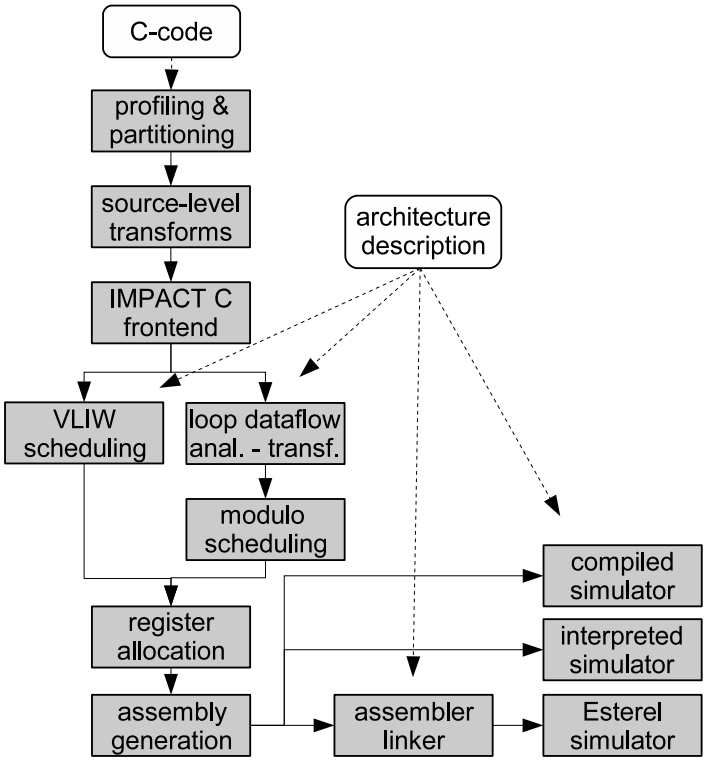


Figure 3. DRESC Compiler Framework

is automatically identified and handled by the DRESC compiler through joint register allocation for the shared RFs. The compiler generates scheduled code for both reconfigurable array and VLIW. The scheduled code can already be simulated by a cycle-true simulator. It is also processed by the assembler and linker to generate binary code. The binary code is used by a low-level Esterel simulator and as final format for running on the hardware.

In the following text, we describe some parts of the tool flow in further details. These parts are important for doing hardware implementation of ADRES architecture and not published previously.

2.4.1. *XML-Based Architecture Description*

The XML-Based architecture description plays a central role in defining an ADRES instance. Unlike other processor architecture description languages (Fauth et al., 1995; Hadjiyiannis et al., 1997; Pees et al., 1999), our XML-based description focuses on only high-level features of an architecture like the amount of resources and their topology, which is the mostly needed information for the compiler. We didn't provide mechanisms to define the semantics of each individual operation. Instead we assume the ADRES architecture will

inherit the operation set generated by the compiler frontend. This assumption simplifies the compiler support. We also simplify the pipeline description by assuming all the FUs are fully pipelined. Therefore, only the latency of each type of operation needs to be specified. The current description language is extensible.

```
<resource>
  <FU name = "fu_0">
    <in name = "pred" width = "1" />
    ...
    <out name = "pred_dst1" width = "1" />
    ...
    <op>
      <opgroup name = "ldmem"/>
      ...
    </op>
  </FU>
  ...
  <RF name = "ireg_10" width = "32" size = "8">
    <in name = "in1" />
    <out name = "out1" />
    <out name = "out2" />
  </RF>
  ...
  <TRN name = "outireg_0" width = "32" delay = "1" />
  ...
</resource>
```

Figure 4. The resource section

The overall architecture description comprises main three sections: *resource*, *connection* and *behaviour*. The *resource* section allocates a number of resources of different types (Figure 4). The resources include FUs, RFs and TRNs (transitory nodes such as muxes and busses). For FUs, the names of input and output ports, data width and supported operation groups can be specified. The operation groups themselves are defined in the *behaviour* section. RFs are specified in a similar way. TRNs currently include output registers and busses because data are passed through these components in a transitory way. Their specification includes data width and delay. Figure 4 shows all three types of resources allocated in the *resource* section. The *connection* section defines the topology of an ADRES instance.

Figure 5 shows some examples of connections. Generally, it connects an output port of an entity to an input port of another entity. If one entity doesn't have a port, e.g., output register, the port field can be skipped. If multiple output ports are connected to one input port, an implicit multiplexor will be created for the input port. The *behaviour* section defines some other architectural properties (Figure 6). For example, it specifies which FUs and RFs are used in the VLIW processor.

The XML file is quite large because each resource and connection have to be individually described to allow maximal flexibility. A script languages such as PHP language can be used to produce XML file in a convenient way.

```
<connection>
 <connect>
  <src entity = "vliw_pred"  port = "out1" />
  <dst entity = "fu_0" port = "pred" />
 </connect>
 <connect>
  <src entity = "fu_0" port = "pred_dst1" />
  <dst entity = "vliw_pred" port = "in1" />
 </connect>
 ...
 <connect>
  <src entity = "outpred_2" />
  <dst entity = "pred_col_bus2" />
 </connect>
 ...
</connection>
```

Figure 5. The connection section

```
<behaviour>
   <vliw_section>
      <vliw_reg name = "vliw_reg" />
      <vliw_pred name = "vliw_pred" />
      <stop_sig name = "loop_stop" />
   </vliw_section>
   <op_section>
    <opgroup name = "arith" delay = "1">
       <op name = "mov" />
       <op name = "abs" />
       <op name = "add" />
       ...
    </opgroup>
    ...
   </op_section>
```

Figure 6. The behaviour section

The architecture description is used in many places of our tool flow, including compiler, simulators and XML2VHDL tool.

2.4.2. *XML2VHDL Tool*

An XML2VHDL tool is developed to produce synthesizable VHDL directly from architecture description. The tool generates the high-level structural VHDL to compose all the basic components together. It also take care of some tedious and error-prone tasks such as encoding of multiplexor selection.

It is not fully automated, however, leaving designer to design and optimize basic components such as FUs and RFs. Other components such as instruction cache and configuration memory are also pre-designed and provided as libraries to XML2VHDL tool. These components are not specified in architecture description currently.

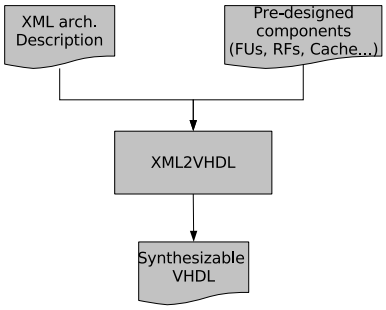


Figure 7. XML2VHDL tool flow

2.4.3. *Verification Support*

To meet various requirements from application development to hardware verification, we developed several simulators. All the simulators can co-simulate both the VLIW and the array parts.

The first simulator belongs to a category of so-called compiled simulators. A compiled application on the target architecture is translated back to the C code that simulates the execution of the assembly/machine code. Hence, the generated C code is compiled by the host compiler to an application-specific simulator running on the host computer. The compiled simulator can do very fast simulation, allowing a programmer to verify his design quickly. It is also provide cycle-true simulation, almost as accurate as HDL simulation (3% margin).

For hardware verification, an Esterel-based simulator has been built. Esterel (Berry and Gonthier, 1992) is both a programming language, dedicated to programming reactive systems, and a compiler which translates the Esterel code into a flattened finite-state machine implemented in C. Our Esterel-based simulator models an ADRES architecture structurally similar to the HDL implementation, but at a higher level. It simulates much faster and emits data traces that are used for hardware verification. We use it extensively during our hardware verification phase and for exploring architectural design options.

We are also developing a new conventional interpretive simulator, which is an independent program taking directly the compiled application as input. It interprets and executes the assembly/machine code at run-time operation-by-operation on the simulated target architecture. A designer can use traditional debugging commands such as *step* and *breakpoint*. It can also do the simulation in different accuracy/speed levels and produce different trace levels for verification. Additionally, this simulator can be used in system-level simulation. For example, it has been integrated with ARM simulator using CoWare tool to perform simulation of a system consisting multiple ARM and ADRES processors.

2.4.4. *Power Simulation Support*

Low-power is an extremely important goal for targeted application of ADRES architecture. Therefore, we build extensive power simulation support in our flow. The power simulation is based on Esterel simulator and third-party tool. The key for power simulation is to obtain as accurate as possible switching activity of the hardware. As mentioned in previous section, Esterel simulator is very close to HDL simulation and runs much faster. Thus, it is extended to producing the toggling file. On the other hand, we use realistic hardware characteristics after synthesis, placement and routing. With all the information, we use a third party tool, i.e., Synopsys Prime Power, to calculate power consumption.

3. Instance Design and Hardware Implementation

With ADRES reconfigurable processor template, we can easily design processor targeting at different application domain. The fully retargetable toolset allow designers to quickly explore different design options.

In this project, our goal is to design a high-performance low-power media processor for video codec, especially for H.264/AVC decoder application. More concretely, the media processor should reach 300 MHz with 90nm standard-cell technology. It should be able to decode D1 (720x480) quality video stream at real-time (30 frames/sec.) and consume no more than 100mw of power. Additionally, we would also like to build a demonstrator based on FPGA. It should at least reach 50MHz and decode QVGA bitstream at real-time.

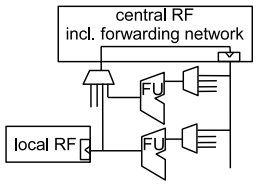
We do not have automatic way to determine optimal size and topology of a CGRA array. We determine array size according to design goal and analysis of applications. With some early synthesis experiment. It became clear very soon that a 4x4 ADRES array was the best choice for H.264/AVC decoder. With larger arrays, the limited connectivity of the FPGA would become problematic. Also, the 4x4 array still achieves relatively high performance because many loops do not contain enough parallelism to utilize all resources of a larger array.

To reach these concrete objectives, two main design exploration tracks were followed: instance design and application mapping. Here we only discuss instance design.

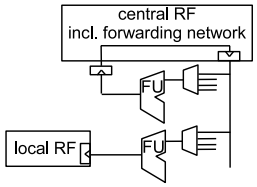
3.1. ADRES Core Exploration

3.1.1. *Architecture Exploration*

Since the ADRES template offers vast design space, to achieve a given design target, we can explore the design space to obtain an optimized instance.



(a) unpipelined, densely-connected path



(b) pipelined, sparsely-connected path

Figure 8. A fragment of our original instance (a), and an adapted version thereof (b).

This optimization step included, amongst others, the insertion of pipelining registers in the reconfigurable array to break critical paths.

For example, our first trial instance included paths as detailed in Figure 3.1.1(a). Note that in paths through the central RF, there occurs only one buffer: at the output of the forwarding network of the RF. Soon we discovered that such long paths would prohibit us from reaching our target frequency. To solve this problem, we adapted the instance as depicted in Figure 3.1.1(b). Basically, we reduced the number of connections going from FU outputs to the input ports of the central RF, thus saving on muxes, and we inserted a pipelining latch to break the critical paths.

While both changes potentially deteriorate performance (because of limited connectivity, and because of longer delays on certain execution paths), the effect of these changes on performance was very limited in practice. Most importantly, this follows from the fact that the additional delays are not on critical paths in the software-pipelined loops because their schedule most-often is resource-bound rather than data-dependence bound. Secondly, the number of live-out variables (variables that see their values changed during a loop's execution, and that need to be stored in the central RF) is limited in most loops, so the connectivity to the write ports of the central RF is also not a critical point.

Another important change that was made during our design space exploration was to move to a 3-issue VLIW machine instead of a 4-issue machine. Because most high-ILP loop code is executed in array mode, it is not necessary to have a very wide VLIW machine. Instead, by reducing the number of VLIW issue slots from 4 to 3, the number of ports on the central register file

is reduced from 12 (8 read, 4 write ports) to 9 (6 read, 3 write ports). This increases the speed of the RF and of its corresponding forwarding network significantly, without hurting performance.

3.1.2. *Media-Oriented Instructions*
Like typical media processors, we enhanced FUs with SIMD instructions for data packing and other operations typically used for video codecs. These instructions are implemented as intrinsics. Table I lists these new instructions.

Table I. Media-Oriented instructions implemented as intrinsics

instructions name	function description
clip1	clip a between -b and b (a, b are operands)
clip2	clip a between 0 and b (a, b are operands)
sh_rnd	shift and round
max	return the bigger one of the two operands
min	return the smaller one of the two operands
modulo	restricted modulo operation (only power of 2)
avg	average value of the two operands
avgu4	average value of 4 packed-bytes
pack2	pack two 16-bit values into one 32-bit word
spacku4	pack four unsigned bytes into one 32-bit word

3.2. PROCESSOR-LEVEL ARCHITECTURE

The ADRES core alone does not work. It needs memory hierarchy to store data and I/O subsystem to get data in and out. The whole processor-level diagram is shown in left part of Figure 3.2. The processor contains an ADRES core, a data RAM, a VLIW instruction cache, and a configuration memory. To reduce power consumption, we use software controlled data memory instead of using data cache. In this media processor, we use 4 banks of SRAM to provide up to 4 simultaneous read/write capability. It is sufficient to a 4x4 array. Each bank is 8 KB of size. To support multiple data accesses to memory, a novel queue-based memory organization is used to reduce potential bank conflicts (Mei et al.,). Many small queues (1 to 3 entries) are attached to each load/store unit. The simultaneous memory accesses to a single bank will be smoothed out by buffering requests in the queues.The queue-based memory

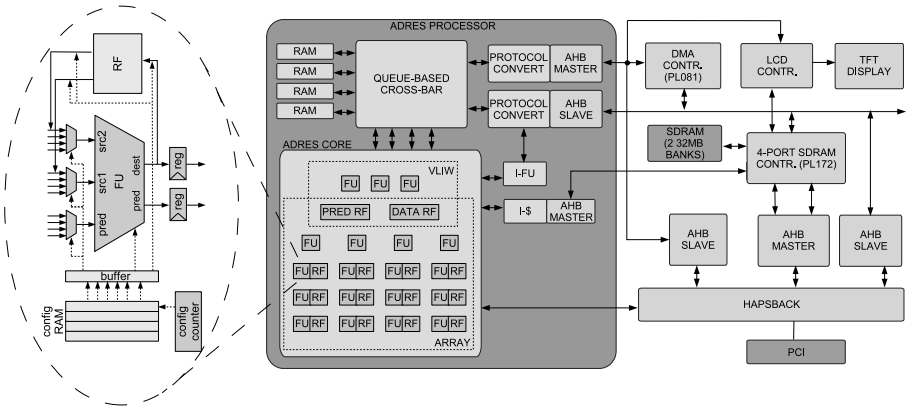


Figure 9. Overview of our target platform. On the left, details of an FU and its local RF are shown.

organization is totally transparent to the compiler. The only difference is that load/store operations have longer latencies (1 or 2 cycles typically) to allow sufficient time for buffering. It brings no major penalty on performance because pipelined loops can hide instruction latency, but reduces bank conflicts significantly. Currently, the memory subsystem can support up to 8 memory banks, which is sufficient for 4x8 or 8x8 array. The VLIW instruction cache is a direct-mapped 32kB cache with a line size of 64 bytes. The configuration memory holds 256 configurations, more than enough for our purposes. Its width depends on the number of configurable elements within the array; in the instance used in this instance each line is 896 bits long, thus totaling the configuration memory size to 27kB.

In the system, these memories in processor are communicated with the main system SDRAM through AMBA AHB bus and an DMA controller. The data swapping between data memory and SDRAM is controlled by software, i.e., the application developer inserts API (application programming interface) calls in the proper places to initiate DMA transfer.

3.3. Synthesis and Physical Design

We optimized the VHDL code generated with our HDL generator. In particular, the implementation of the FUs and RFs in our ADRES core were optimized in order to achieve higher clock and smaller area. We also apply other performance and power optimizations such as clock-gating and isolating switching activity.

Figure 10 depicts the hardware implementation of the media processor. It is able to achieve clock speed of 300MHz. Considering that we are able to deliver more IPC than the state-of-art DSPs and the design still lacks fine

Table II. Tools and libraries used for synthesis and physical design

Synthesis tool	Synopsys Physical Compiler 2006.06
Place and route tool	SoC Encounter v05.20
Simulation tool	ModelSim v6.0a
Power tool	Synopsys Prime Power ver. X-2005.06-SP2
Standard cell library	TSMC 90nm
Configuration and Data Memories	Artisan TSMC nominal Vt Single Port RFs and SRAMs

optimization in circuit- and physical-levels, we believe that our results are quite competitive. The die size is only $3.6mm^2$ though the media processor has 4x4 FUs. It is comparable with even a RISC processor. For example, an ARM Cortex-A8 processor occupies about $4mm^2$ with 65nm technology. One reason is that memory is typically dominating part in a modern embedded processor.

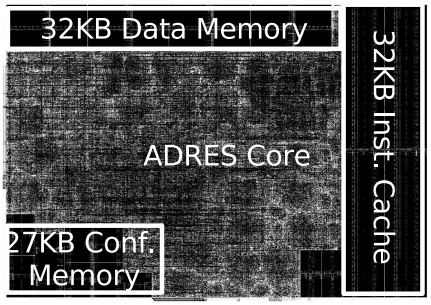


Figure 10. Hardware implementation of the media processor: die size = $3.6mm^2$; clock speed (worst case) = 300MHz

3.4. FPGA-BASED DEMONSTRATOR

We have yet made tape-out of the media processor. Instead, the whole design and tool flow are proved on an FPGA-based demonstrator platform. Figure 3.2 depicts the entire platform we implemented on FPGA, including the media processor itself and other storage, I/O and controller components. The demonstrator platform is implemented on a HARDI HAPS-32 prototyping board equipped with two Xilinx Virtex-4 LX200, a PCI interfacing board, a 32MB SDRAM board, and an interconnect board to enhance the connectivity between the two FPGAs. One FPGA houses the processor, while the other houses the rest of the platform. We use a synchronous clock for the system:

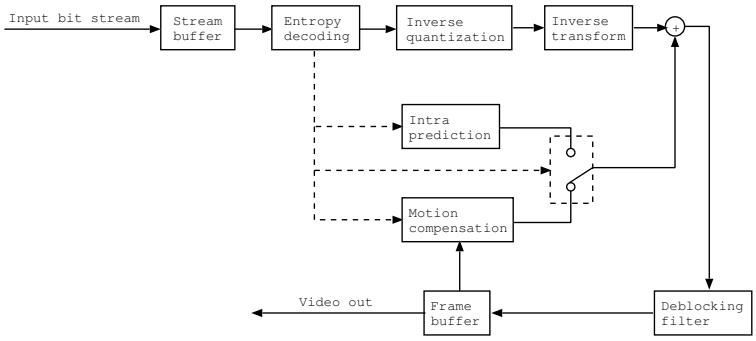


Figure 11. Scheme of the H.264 decoder.

50MHz for the processor and 25MHz for the peripherals. Synplify Premier was the only tool which reached the synthesis target for the processor. The utilization of the processor FPGA is 9% of DSP48, 34% of IOBs, 94% of BRAMs and 43% of slices. The utilization of the other FPGA is 0% of DSP48, 46% of IOBs, 43% of BRAMs and 15% of slices.

4. Performance and Power Evaluation

We mapped an H.264/AVC decoder application onto the media processor to evaluate its performance and power consumption.

4.1. H.264/AVC Decoder Overview

The ITU-T H.264, also known as MPEG-4 (Part 10) Advanced Video Coding represents the latest evolution of video codecs (Wiegand et al., 2003). Like with every video codec evolution step, encoding quality for a given bit rate is improved, at the expense of computational complexity. For the baseline profile, the computational complexity lies mainly in two components: The deblocking filter, which is now part of the decoding loop, and the motion compensation with quarter-pixel interpolation, for various block sizes ranging between 16x16 and 4x4. The many variations in macroblock-encoding increase the amount of control code involved in the processing. Looking at the decoder from the perspective of decoding a sequence of macroblocks, the processing flow becomes mostly non-uniform.

The functional block diagram of a generic hybrid video decoder based on the H.264 standard is shown in Figure 4.1. The incoming video bitstream is stored in a memory buffer in order to be parsed and decoded by the entropy decoding stage. The syntax elements obtained after this process for each macroblock are demultiplexed and sent to the different functional kernels

involved in the decoding process. In particular, the syntax elements related with the coding of the luminance and chrominance residual samples of the current macroblock (MB) are re-ordered by following a typical inverse scan procedure and passed to the inverse transform kernel. In parallel, a predictor is composed from previously decoded pixels in the same frame (intra coded macroblocks) or from pixels pointed by the received motion vectors belonging to frames previously decoded (inter coded macroblocks) depending on the information stored in the MB layer of the received bitstream. By adding the inverse transformed residual samples to the predictor selected and filtering the result in order to reduce the presence of annoying blocking artifacts, the original macroblock is recovered with minimal quality losses.

Important kernels for acceleration are in the motion compensation, in the deblocking filter, the inverse transform, and some data handling functions like memset. The following discusses the implementation of the decoder.

4.2. Code Preparation

For availability and performance reasons we decided to use the H.264 decoder implemented in the libavcodec library (FFMPEG,), and not the reference decoder. To be able to embed the code, we first extracted the H.264 decoder from the libavcodec sources and wrote a simple decoding loop around it, and then pruned unused functions from the sources.

For our target system some further code cleaning had to be done: Floating-point code had to be removed, as well as 64 bit and unaligned memory accesses. Furthermore, the code must comply with the older ANSI C standard (C89) because of the IMPACT front-end version (2.36) we are currently using. The target hardware and the compiler do not support modulo or division operations, therefore these had to be replaced as well. Finally, the ADRES implementation has only 16 bit multipliers, which required some additional analysis and code changes. Finally we implemented some needed C library functions like memcpy and memset and replaced dynamic with static memory allocation.

4.3. Mapping onto the Media Processor

Because an ADRES processor executes in two modes, the VLIW mode and the array mode, the application has to be partitioned and both parts have to be tailored for their execution mode. This requires manual adaptations to the C code of the application, that are discussed here. We should note, however, that only standard C code is used. There is no inline assembly, nor are there pragmas in the code.

First, an ADRES software developer needs to partition his code in those procedures in which loops will be mapped to array mode, and in the remaining procedures that will be mapped onto VLIW mode only. The former are

identified by a procedure name that starts with the prefix DRESC. Although this partitioning is not strictly required, it is beneficial for the quality of the generated code: because transformations that enable/improve the mapping of loops onto array mode are not necessarily beneficial for code that will be mapped onto VLIW mode, we have adapted the IMPACT C-frontend to treat both types of procedures differently during its optimizations. Hyperblock formation, for example, is applied less aggressively on code that will not be mapped onto array mode. Vice versa, inlining is applied more aggressively on code that will be mapped onto array mode.

Next, the loops that will be mapped onto array mode often need some preparation to make them better suited for mapping them onto the array. Typical transformations that are applied for this purpose are *loop coalescing* and *loop unrolling*. Loop coalescing is the combination of two nested loops in one single loop. In the coalesced loop, the number of iterations is bigger than in the original inner loop, and hence switches between VLIW and array modes can be saved, as well as executing code of the outer loop in the less efficient VLIW mode. Loop unrolling typically increases the number of operations per iteration. Thus, the compiler has more operations available to schedule in the array mode, thus allowing him to exploit the offered ILP optimally.

To select the code to be mapped onto array mode, we used profiling information. First, we identified the hot code, and once we had identified this code, we studied it in detail to see which loops were potential candidates to be executed in array mode. In total, 28 out of 222 procedures were chosen to have their loops mapped onto array mode, for a total of 50 loops. 6 loops originate from the deblocking filter, 31 originate from the motion compensation, 5 from the inverse transformation, 3 from the decoder control part, and finally 5 from memory copy and initialization operations. The remaining procedures, including entropy decoder, are mapped onto VLIW-only mode. These procedures are either too control-insensitive to be pipelined, or simply not loops.

4.4. Performance Evaluation

Several input streams have been decoded on our implementation to measure its performance on the hardware board. In order of increasing complexity these are the well-known News,Mobile, and Foreman sequences. For each sequence, we decoded two image sizes,QVGA (320x240) and CIF (352x288), at two bit rates per image size (128kb and 256kb). Furthermore, each combination was measured on a program version that was compiled to VLIW mode only, and on a version that was compiled to both VLIW and array mode. The results of our measurements are presented in Table 4.4. The instructions-per-cycle, the framerates and the required frequency to obtain real-time decoding are all measured on the hardware, i.e., on the host system connected to the

Table III. For movies of increasing complexity, different sizes and different bitrates, the number of executed instructions per cycle (IPC) is presented, the framerates when playing at 50MHz, both in VLIW mode and in mixed VLIW-array mode, and the required frequencies for playing the movies in real time. Also included is the fraction with which the simulator (excluding stall cycles) overestimated the performance on the hardware (including I-cache stalls and memory conflict stalls).

sequence	format	bitrate (Kb)	VLIW mode only				VLIW and array mode			
			IPC	framerate (fps at 50 MHz)	RT freq. (MHz at 30fps)	HW cycles / sim. cycles	IPC	framerate (fps at 50 MHz)	RT freq. (MHz at 30 fps)	HW cycles / sim. cycles
news	QVGA	128	1.83	23.78	63.1	1.07	4.74	51.24	29.3	1.10
news	QVGA	256	1.81	19.39	77.4	1.06	4.58	41.01	36.6	1.10
news	CIF	128	1.84	19.34	77.6	1.06	4.83	41.85	35.9	1.09
news	CIF	256	1.82	16.05	93.5	1.06	4.70	34.47	43.5	1.10
mobile	QVGA	128	1.99	16.54	90.7	1.09	5.90	42.99	34.9	1.11
mobile	QVGA	256	1.95	13.23	113.4	1.08	5.64	33.28	45.1	1.11
mobile	CIF	128	1.97	14.39	104.2	1.08	5.81	37.12	40.4	1.10
mobile	CIF	256	1.97	11.13	134.8	1.07	5.80	28.27	53.1	1.11
foreman	QVGA	128	1.98	14.73	101.8	1.09	6.05	39.02	38.4	1.13
foreman	QVGA	256	1.94	12.05	124.6	1.09	5.82	31.02	48.4	1.13
foreman	CIF	128	1.99	12.02	124.8	1.09	6.06	31.98	46.9	1.12
foreman	CIF	256	1.96	9.97	150.5	1.09	5.97	26.37	56.9	1.12

FPGA board. These numbers hence include the impact of instruction cache misses and of SDRAM arbitration. All results presented are averages over 300 frames (10 seconds).

As can be seen in the table, our implementation achieves real-time decoding for all but two of the sequences. The operating frequency of 50 MHz would have to increase by only 6.9 MHz in order to achieve real-time decoding for all the streams. By comparing the VLIW-only and the mixed VLIW-array numbers, it is clear that ADRES's array mode is indeed an excellent accelerator.

To indicate how competitive these results are, we mention that an H.264 baseline decoder implemented for the StarCore 2400 processor in manually optimized assembly code requires 64MHz for real-time CIF (StarCore-Hantroh264, 2006). So not only does the ADRES architecture *per se* outperform the StarCore architecture, we now have also demonstrated that a C-programmed implementation on ADRES outperforms an assembly-level optimized implementation on StarCore.

Besides measuring the real performance of our system, we also did performance measurements using the compiled-code simulator of the ADRES core. As this excludes cache misses, data memory arbitration and SDRAM

arbitration, it represents an upper bound for the achievable performance. We see that the results differ by a relatively small margin of 6 to 13%, which demonstrates a good system design. The difference is bigger when code is mapped to array mode. This follows from the fact that not only the overall IPC is higher in array mode, but so is the number of data memory accesses per cycle. Consequently, the relative number of stalls due to data memory arbitration are higher as well. Also noticeable is that the framerate improvement obtained by using the array mode is a little bit less than the IPC improvement, resulting from the fact that in array mode more code is executed speculatively than in VLIW mode.

4.5. Other Applications

To show the effectiveness of the DRESC compiler, we've compiled three benchmark programs on two architecture instances of ADRES. One is the standard instance used throughout this paper. The other instance is obtained by inserting extra pipeline registers in the standard instance as discussed in Section 3.1.1.

The MPEG-4 video decoder (AVC H.264) and an MPEG-4 encoder (MPEG4-enc) are two benchmarks from the multimedia domain and a multiple antenna receiver (MIMO) applications has been chosen from the wireless domain. The results are presented in Table IV. When an application consists of more than ten loops, only the ten most frequently executed ones are included in the table.

The resource-constrained minimal II (ResMII) is calculated by totaling, for each resource, requirements imposed by one iteration of the loop. The recurrence-constrained minimal II (RecMII) corresponds to the sum of the latencies divided by the sum of dependence distances along the most critical cycle (a.k.a loop-carried dependencies) in the dependence graph. Readers are referred to (Ramakrishna, 1994) on how to compute ResMII/RecMII. The MII is the larger one of ResMII and RecMII. The II column shows the effectively obtained II after compilation. Since both instances have 16 FUs, the maximum IPC (number of Instructions Per Cycle) is 16. The time column shows the time (in seconds) to compile the loop.

We can draw the following conclusions from the results in Table IV. First, the average IPC of the evaluated loops is over 10 for each application. For the resource-bound (RecMII $\leq$ ResMII) loops, the obtained IPC is even higher, reaching 13.7 for one loop. The 6 recurrence-bound loops (RecMII $\geq$ ResMII) cannot take advantage of the high number of parallel resources. Still, IIs are obtained for these loops that are close to RecMII. For 18 out of the 19 loops our algorithm finds a schedule at an II which equals MII or MII+1 when compiling for the pipelined processor versions.

Secondly, because of the pipelining registers, the pipelined VLIW versions have higher RecMII than the non-pipelined versions. Thus, it is not

surprising that the recurrence-bound loops are scheduled with lower IIs on the non-pipelined version. The fact that the compiler is not hampered by the pipelining registers in the resource-bound loops demonstrates the effectiveness with which the compiler can work around the additional latencies that one expects with deeper pipelines. This illustrates how the scheduler uses the pipelining registers to its advantage as temporary storage locations.

4.6. Power Simulation Results

Since low-power is an essential design goal of the ADRES architecture and its targeted application domain, we integrate extensive power simulation support in our tool flow (see Section 2). Table V lists the simulated dynamic power and static power of the media processor running at 300MHz. Since we can decode CIF format at 50MHz, 300MHz should be more than sufficient for a D1 format video. The total power of 105.5mw is quite competitive. To put this power figure into perspective, TI's ultra low-power C5000 DSPs consumes about 0.4 mW/MHz (1.2 v * 0.33 mA/MHz) (TI, 2007). Hence, it would consume roughly 120mW at 300MHz, which is comparable to our media processor. However, our media processor is much more powerful than C5000 since it can exploit much higher parallelism (16 FUs vs 2 FUs). To perform the same AVC decoding task would require TI's high-end DSPs, e.g., C64+ series.

Figure 12 shows power breakdown of its different components. Clock tree, instruction cache, data memory and ADRES core roughly each accounts for 1/4 of total power consumption. Though ADRES architecture uses much more resources, mainly FUs and RFs, than normal processors, it doesn't add much to the total power consumption. Clock tree and memories dominates the total power consumption, not the ADRES core itself.

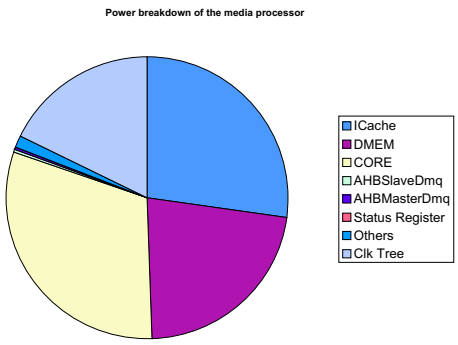


Figure 12. Power distribution of the media processor

5. Related Work

According to R. Hartenstein, CGRA refers to a class of reconfigurable architectures that provide operator level basic configurable block, word level datapaths as well as powerful and very area-efficient routing switches (Hartenstein, 2001). Many CGRAs have been proposed in recent years. MorphoSys is a typical coarse-grained reconfigurable array, consisting of 8x8 basic units, called reconfigurable cells (RC) (Singh et al., 2000). Each RC consists of an ALU, a multiplier and a register file. The interconnection is through multiplexors and busses. The MorphoSys array cannot work alone for an entire application, so it is coupled with a RISC processor, called TinyRISC, to form a reconfigurable system. A frame buffer together with a DMA controller provides the communication channel between the two entities. The mapping of kernels is based on an assembly-like language or a GUI-based capture tool (Mo04,), where the designer does all the scheduling work manually. In contrast to other 2-dimensional architectures, PipeRench features a ring-like one-dimensional architecture (Goldstein et al., 2000), specially designed for pipelining. The architecture is organized as stripes of processing elements (PEs), which are equivalent to FUs. A number of stripes are connected by an interconnection network in a ring way in order to provide pipelining capability. Compilation techniques are developed to automatically map kernels onto a PipeRench fabric based on placement and routing (Budiu and Goldstein, 1999). High-level languages like C and Java are partially supported. The PipeRench architecture is usually used as a co-processor. It has to be combined with a processor to execute an entire application, but how to co-design for such system is not well solved. RaPiD is a coarse-grained architecture developed in the University of Washington (Ebeling et al., 1996). Like PipeRench, RaPiD is also a one-dimensional architecture designed to run pipelined dataflow. The RaPiD is an architecture template that should be customized to an application rather than a generic reconfigurable architecture. The functional units in the datapath provide the operations that are performed by the target applications on the Rapid array. A C-like language, called RaPiD-C, accompanied with a compiler, is developed to program the RaPiD architecture (Cronquist et al., 1998). The linear structure of the RaPiD array greatly reduces the implementation and compilation complexity. Silicon Hive is a spin-off company from Philips Electronics (SiliconHive, 2006). Its architecture consists of three levels. The basic block is PSE (processing and storage element), which is essentially a partially connected VLIW processor. Several PSEs together form a Cell. It is still fully synchronous because there is only one controller within the Cell. PSEs are communicating through CL (communication lines). In the Cell level, a dataflow can be mapped to PSEs to exploit high parallelism. The core design tool of Silicon Hive is the HiveCC spatial compiler. It accepts a program written in a sub-set of ANSI C and

uses constraint-analysis and scheduling techniques to pipeline the loop. The constraint analysis module is able to handle the partial connectivity of the architecture. Readers are refered to (Hartenstein, 2001) for more CGRAs.

At the best of our knowledge, none of these architectures have provided full toolset support with automatic compilation and proved with complex application such as H.264/AVC on real hardware platform.

6. Conclusions and Future Work

An in-depth overview of the ADRES architecture template has been presented, including its operation mode and its toolset based on the DRESC ANSI-C compiler. This toolset has become mature enough to build a hardware platform based on a specific media processor based on the ADRES template.

Using the media processor discussed in this paper, we have been able to demonstrate that the toolset can handle complex applications such as an H.264/AVC decoder, and that it enables us to obtain very competitive results both with respect to the hardware implementation and its clock speed, and with respect to the software compilation and the obtained performance. The media processor is able to do C-programmed real-time H.264/AVC CIF decoding at 50 MHz, and it only consumes 105 mW at 300MHz. Thus, we have demonstrated the efficiency of the ADRES architecture, and the effectiveness of its compiler.

Future work for ADRES and DRESC includes possible tape-out of the media processor, for which we first want to optimize the hardware implementation in further. This optimization will include techniques such as operand isolation, clock gating, etc. With respect to the toolset, we plan to improve the efficiency of the compiler by using improved, significantly faster placement and routing algorithms, and to look for alternatives for the simulated annealing approach that is currently used.

References

Allen, J. R., K. Kennedy, C. Porterfield, and J. Warren: 1983, 'Conversion of Control Dependence to Data Dependence'. In: *Proc. of ACM Symposium on Principles of Programming Languages*. pp. 177–189.

Berry, G. and G. Gonthier: 1992, 'The Esterel Synchronous Programming Language: Design, Semantics, Implementation'. *Science of Computer Programming* **19**(2), 87–152.

Budiu, M. and S. C. Goldstein: 1999, 'Fast Compilation for Pipelined Reconfigurable Fabrics'. In: *Proc. of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. pp. 195–205.

Cronquist, D. C., P. Franklin, S. G. Berg, and C. Ebeling: 1998, 'Specifying and Compiling Applicaitons for RaPiD'. In: *Proc. of Field-Programmable Custom Computing Machines (FCCM)*. pp. 116–125.

Ebeling, C., D. Cronquist, and P. Franklin: 1996, 'RaPiD - Reconfigurable Pipelined Datapath'. In: *Proc. of International Workshop on Field Programmable Logic and Applications*. pp. 126–135.

Fauth, A., J. V. Praet, and M. Freericks: 1995, 'Describing Instruction Set Processors Using nML'. In: *Proc. of Design Automation Conference (DAC)*. pp. 503–507.

FFMPEG, 'libavcodec is distributed with FFmpeg, an audio/video conversion tool'. http://sourceforge.net/projects/ffmpeg.

Goldstein, S. C., H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor: 2000, 'PipeRench: A Reconfigurable Architecture and Compiler'. *IEEE Computer* **33**(4), 70–77.

Hadjiyiannis, G., S. Hanono, and S. Devadas: 1997, 'ISDL: An Instruction Set Description Language for Retargetability'. In: *Proc. of Design Automation Conference (DAC)*. pp. 299–302.

Hartenstein, R.: 2001, 'A Decade of Reconfigurable Computing: a Visionary Retrospective'. In: *Proc. of Design, Automation and Test in Europe (DATE)*. pp. 642–649.

IMPACT, 'The IMPACT Group'. http://www.crhc.uiuc.edu/impact.

Maestre, R., F. J. Kurdahi, M. Fernndez, R. Hermida, N. Bagherzadeh, and H. Singh: 2001, 'A Framework for Reconfigurable Computing: Task Scheduling and Context Management'. *IEEE Trans. on VLSI Systems* **9**(6), 858–873.

Mei, B., S. J. Kim, and R. Pasko, 'A new multi-bank memory organization to reduce bank conflicts in coarse-grained reconfigurable architectures'. In: *sumbitted to IEEE Transaction on VLSI Systems*.

Mei, B., S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins: 2002, 'DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures'. In: *Proc. of International Conference on Field Programmable Technology*. pp. 166–173.

Mei, B., S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins: 2003a, 'ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix'. In: *Field-Programmable Logic and Applications*.

Mei, B., S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins: 2003b, 'Exploiting Loop-Level Parallelism for Coarse-Grained Reconfigurable Architecture Using Modulo Scheduling'. *IEE Proceedings: Computer and Digital Techniques* **150**(5).

Mo04. http://www.eng.uci.edu/morphosys/tools.html.

Pees, S., A. Hoffmann, V. Zivojnovic, and H. Meyr: 1999, 'LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures'. In: *Proc. of Design Automation Conference (DAC)*. pp. 933–938.

Ramakrishna, R. B.: 1994, 'Iterative Modulo Scheduling'. Technical report, HPL-94-115, HP Labs Technical Reports.

Rau, B. R., M. Lee, P. p. Tirumalai, and M. S. Schlansker: 1992, 'Register Allocation for Software Pipelined Loops'. In: *Proc. of ACM SIGPLAN Conf. Programming Language Design and Implementation*. pp. 283–299.

SiliconHive: 2006, 'SiliconHive'. http://www.silicon-hive.com.

Singh, H., M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho: 2000, 'MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications'. *IEEE Trans. on Computers* **49**(5), 465–481.

StarCoreHantroh264: 2006, 'StarCore and Hantro announce immediate availability of high-performance H.264 video decoder'. Press Release, Feb. 13, 2006.

TI: 2007, 'Texas Instruments'. http://www.ti.com.

Wiegand, T., G. J. Sullivan, G. Bjontegaard, and A. Luthra: 2003, 'Overview of the H.264/AVC Video Coding Standard'. *IEEE Trans. on Circuits and Systems for Video Technology* **13**(7), 560–576.

Table IV. Results for the benchmark loops. First, the target-version-independent number of operations (#ops) and the ResMII. Then for each target version the RecMII, the actually achieved II and IPC, and the compilation time.

			pipelined				non-pipelined			
Loop	#ops	Res MII	Rec MII	II	IPC	time (sec)	Rec MII	II	IPC	time (sec)
AVC										
MBFilter1	70	5	2	6	11.7	251	1	6	11.7	291
MBFilter2	89	6	7	9	9.9	529	6	8	11.1	433
MBFilter3	40	3	3	4	10.0	112	2	3	13.3	110
MBFilter4	105	7	2	9	11.7	538	1	9	11.7	599
MotionComp	109	7	3	10	10.9	183	2	10	10.9	213
FindFrameEnd	27	4	7	7	3.9	26	6	6	4.5	40
IDCT1	60	4	2	5	12.0	179	1	5	12.0	191
MBFilter5	87	6	3	7	12.4	399	2	7	12.4	309
Memset	10	2	2	2	5.0	8	1	2	5.0	11
IDCT2	38	3	2	3	12.7	99	1	3	12.7	72
Average					**10.0**				**10.5**	
MPEG4-enc										
MotionEst1	75	5	2	6	12.5	88	1	6	12.5	135
MotionEst2	72	5	3	6	12.0	115	2	6	12.0	169
TextureCod1	73	5	7	7	10.4	188	6	5	12.2	186
CalcMBSAD	60	4	2	5	12.0	89	1	6	12.0	125
TextureCod2	9	1	2	2	4.5	6	1	2	4.5	15
TextureCod3	91	6	2	7	13.0	195	1	7	13.0	132
TextureCod4	91	6	2	7	13.0	194	1	7	13.0	133
TextureCod5	82	6	2	6	13.7	440	1	6	13.7	121
TextureCod6	91	6	2	7	13.0	245	1	7	13.0	175
MotionEst3	52	4	3	4	13.0	46	2	5	10.4	100
Average					**11.7**				**11.6**	
MIMO										
Channel2	166	11	3	14	11.9	162	1	14	10.4	161
Channel1	83	6	3	8	10.4	157	1	8	10.7	79
SNR	75	5	4	6	12.5	89	2	6	12.5	88
Average					**11.6**				**11.2**	

Table V. Power simulation results at 300MHz

total dynamic power	100.7 mW
total leakage power	4.8 mW
total power	105.5 mW