Hardware and a Tool Chain for ADRES

Bjorn De Sutter[1], Bingfeng Mei[1], Andrei Bartic[1], Tom Vander Aa[1], Mladen Berekovic[1], Jean-Yves Mignolet[1], Kris Croes[1], Paul Coene[1], Miro Cupac[1], Aïssa Couvreur[1], Andy Folens[1], Steven Dupont[1], Bert Van Thielen[1], Andreas Kanstein[2] Hong-Seok Kim[3], and Suk Jin Kim[3]

[1] IMEC vzw, Belgium
desutter@imec.be
[2] Freescale Semiconducteurs France SAS, France
a.kanstein@freescale.com
[3] Samsung Advanced Institute of Technology
hong-seok.kim@samsung.com

Abstract. Until recently, only a compiler and a high-level simulator of the reconfigurable architecture ADRES existed. This paper focuses on the problems that needed to be solved when moving from a software-only view on the architecture to a real hardware implementation, as well as on the verification process of all involved tools.

1 Introduction

A new class of programmable processor architectures for embedded applications is emerging: coarse-grained reconfigurable architectures (CGRAs) [3]. Due to difficult programming models and a vast overuse of resources compared to DSP processors, none of them have yet been widely adopted. ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) and DRESC (Dynamically Reconfigurable Embedded System Compiler) try to overcome these issues [6,5].

An ADRES instance consists of an array of basic components, including FUs, register files (RFs) and routing resources (wires, muxes and busses), of which the top row can operate in a VLIW processor mode. This mode shares the central RF with the second mode, the array mode, in which all entities of the architecture operate in parallel. By providing two functional views on the same physical resources, ADRES tightly couples a VLIW processor, which offers an (on other published CGRAs absent) easy path for mapping complex code, and a coarse-grained array that offers unprecedented loop acceleration. The shared central RF minimizes communication and mode-switching costs and enables the compiler to seamlessly generate code for both modes, including data transfers.

In array mode, a loop is executed that consists of the instructions between two configuration memory addresses. To remove control flow from loops, the FUs support predicated execution. The result of each FU can be written to local RFs, which are smaller and have less ports than the shared RF, or routed directly to the inputs of other FUs. The multiplexers in the array are used for routing

K. Bertels, J.M.P. Cardoso, and S. Vassiliadis (Eds.): ARC 2006, LNCS 3985, pp. 425–430, 2006.

data from different sources. The configuration memory stores the configuration contexts, from which a single context is loaded onto the FUs and muxes on a cycle-by-cycle basis, thus in effect reconfiguring them every single cycle.

ADRES is programmed in ANSI C. Our tool chain until recently consisted of three tools. First, the IMPACT C compiler front-end parses C code, optimizes it and generates assembly code in the Lcode instruction format. Next, our DRESC compiler takes an ADRES instance description and maps the loop kernels onto the array [5]. The generated schedules are then simulated with the simulator.

ADRES was designed from a compiler perspective to achieve high-performance, low-power computing with automated C compilation. Until recently, however, ADRES was a virtual architecture for which only the compiler and the high-level simulator existed. Here, we take an important step as we now have a full tool chain that compiles C code and generates binary configuration files that can be executed on a hardware implementation in VHDL. *In concreto*, we have compiled an MPEG2 decoder written in C and we have executed it on an FPGA-based demonstrator, thus proving the concept of ADRES. This short paper focuses on the efforts needed to move from a software-only perspective to a hardware implementation and a full tool chain. Performance results are discussed in other work [5,6]. For a discussion of related work, we refer to Bingfeng Mei's thesis [4].

2 Hardware-Software Interface Issues

Instruction Set. To execute an MPEG decoder on an FPGA implementation of ADRES, without writing a full compiler, the instruction set architecture (ISA) needs to meet three requirements. (1) The VLIW ISA needs to be dense enough to store the program on the limited amount of FPGA memory. (2) For proving to be a potential low-power solution, the decoding of the instructions needs to be simple. (3) The ISA should be close enough to that of the intermediate code representation generated by Lcode representation of the IMPACT compiler. Unfortunately, however, Lcode is much too expressive to be dense. First, Lcode supports a wide range of predicate generating instructions, as available on Intel's EPIC architecture. Most of them are rarely used, so supporting all of them implies a large code size overhead. Secondly, Lcode allows all operands of instructions to be 32-bit immediates. Clearly, this is not feasible in any real ISA.

As a temporary solution, we decided to support only the most common type of *unconditional predicates* [1]. Our tool detects when other types are present in the Lcode, and informs the programmer that he must rewrite his code to avoid the use of that predicate. For rather simple applications, such as an MPEG2 decoder, this solution proved to be satisfactory. To support more complex code, however, such as the H.264 codec, adapting the compiler to generate only the supported predicates is the only feasible solution. Furthermore, ADRES VLIW instructions can have only one immediate operand. When Lcode instructions have more of them, our assembler inserts instructions that first put them in free registers. Because the insertion of additional instructions happens after the code is scheduled, the final schedules are sometimes far from optimal. In a production

tool chain, this workaround solution will need to be replaced by a better instruction selector and post-pass scheduler. Furthermore, our assembler cannot insert instructions in array schedules, because these are too complex to change in a post-pass tool. Instead, we again notify the programmer to change his source code in such a way that constants occurring in CGA-mode loops are first put in temporary variables, and hence in registers instead of in immediate operands. This required only modest source code transformations. Future versions of our tool chain will perform this rather trivial transformation automatically.

Compiler Perspective. We encountered several unexpected problems during the combined development of both the VHDL specification and the binary code format of our architecture. Most often, these problems surfaced because the compiler developers that developed the compiler-supported architecture, approached it from the software perspective, and not from a hardware perspective. In the former, undefined behavior is non-existent in the sense that only specified computations are performed. For example, a nop instruction executed on some FU is supposed not to change the processor state, and is hence given little consideration. In hardware, by contrast, all signals that can influence the program state need to be specified correctly. This impedance mismatch between the two perspectives became apparent when different simulators were developed. Until recently, the only available simulator was a compiled simulator that basically is an RTL-level implementation of the program in C. This C code is then compiled with a standard C compiler, and executed to simulate the program's execution on ADRES. One of the main differences between real hardware, and the compiled simulator, is that the simulator operates on virtual registers that are modeled through variables. Rotating registers are implemented with simple copy operations on the live variables. Consider the following loop on the left hand side:

```
for (i=0;i<10;i++) {
  ... // loop body
}
```

```
i_2 = 0;
while (cont = (i_2<10) || epilogue_is_running) {
   if (cont) i_1 = i_2 + 1 ;
   ... // software-pipelined loop body
   i_2 = i_1; // register rotation
}
```

As the loop index is a loop-carried variable, it is allocated to (virtual) rotating registers, as depicted on the right hand side above. The variable `cont` models the loop continuation predicate, which becomes `false` during the final iteration of the loop, thus informing the processor that the array mode needs to be exited. Because loops are software pipelined, it might still be necessary to continue the loop execution for some cycles, however, in order to finish the execution of the ongoing iterations in the loop epilogue. Until recently, we assumed that the continuation predicate could be set during any stage of the loop: the value of `cont` in the above fragment only depends on the initial value in `i_2`.

During the verification of the hardware, we noted that the above assumption on the `cont` predicate was in fact invalid. In real hardware, not only the live registers rotate, but the other ones do so as well, as shown in the following fragment, that contains two copy statements to mimic register rotation.

```
i_2 = 0; cont = true;
while ((cont && cont = i_2<10) || epilogue_still_running) {
  if (cont) i_1 = i_2 + 1 ;    // loop body
  ...
  i_2 = i_1; i_1 = i_0;       // register rotation
}
```

During the normal operation of the loop, the undefined value of `i_0` that is rotated into `i_1` is overwritten by the guarded increment operation. During the epilogue, however, this is no longer the case, and hence the undefined value in `i_0` is not only copied into `i_1`, but one iteration later also into `i_2`. As a result, `cont` becomes `true` again, and the loop keeps executing.

When we discovered that our assumption was wrong, we were too close to our deadline to start changing the compiler and to not rely on the incorrect assumption. Instead, we opted for slightly adapting the way we used the loop continuation predicate. In the adapted C-code, the continuation predicate is only evaluated as long as it has been true, as can be seen from the while statement in the above fragment. In later versions of our tool chain we will of course adapt the compiler to generate the continuation predicate only at correct schedule times.

While the loop continuation problem is only one issue that arose during this design project, it is very typical for our approach from a compiler perspective. While this approach guaranteed from the start that a full-fledged C-compiler will exist for ADRES, it also resulted in a number of unanticipated problems, like the above one. Consequently, the specification of the architecture needed several updates during its implementation in hardware, because of which a significant amount of time was lost. The lesson we learned is that the hardware and software people involved should learn to speak the same language upfront, and that they should understand the issues involved on both sides of the system.

3 Verification

Several components of our now complete tool chain and hardware implementation needed verification. While the compiler and the compiled simulator had been used and tested for a long period, these tools were largely developed by one PhD. student. [4]. It was expected that there would be hidden assumptions about the compiler in the simulator and vice versa that might need to be reevaluated. Furthermore, the assembler and the linker, that map the assembly-like program representation generated by DRESC onto binary object code, needed to be validated. Finally, the VHDL implementation needed verification. To support these verification needs, a number of tracing tools were developed or extended.

Symbolic Tracing. First, the compiled simulator was extended with tracing capabilities to dump traces containing in and outputs of FUs, RFs and memory accesses. It should be understood that this compiled simulator is just a simple implementation in C of the RTL-like ADRES operations that where generated by the DRESC compiler. In this implementation, local variables of the original program have been replaced by variables that model virtual registers at the RTL

level. Data structures that reside in memory in the original program still reside in memory in the compiled simulator. In fact, the data structure declarations are simply copied from the original program to the compiled simulator. As such, the memory accesses on the compiled program all happen in the address space of the compiled simulator on the host system, and not in the memory space of the binary ADRES program as it would be assembled and linked. In practice, this means that all addresses occurring the RF and FU traces not only depend on the actual program being traces, but also on how the simulator was compiled. After every change to the simulator, the addresses change.

Cycle-Accurate Tracing. Secondly, we developed a cycle-accurate μ-arch simulator by means of Esterel [2]. This simulator simulates binary ADRES executables at the RTL-level at a lower level of abstraction than the compiled simulator. This cycle-accurate simulator thus offers the same tracing capabilities as the compiled simulator, but it's traces are closer to the actual hardware.

The main goal of the cycle-accurate tracing is the verification of the compiler back-end and of the correct operation of the whole ADRES concept. For that purpose, its traces could be compared to those produced with the trusted compiled simulator, thus inheriting the thrust from it. Unlike the compiled simulator, however, the μ-arch simulator runs the program in the ADRES address space. Hence addresses occurring in the traces of both simulators are different, which complicates the comparison of their traces.

A first workaround involves eliminating addresses from the traces. This can be done by only tracing instructions that do not usually operate on absolute addresses, such as multiplications, shifting, or the loading/storing of bytes and words. An alternative is to use two versions of the compiled simulator that executes on different addresses. Then we can first compare the traces of those versions, and detect (and later neglect) the values that are in fact addresses, as these are exactly those values that are different in the two versions. A final alternative is the replacement of addresses in traces by symbol names. This can only be done for data whose symbol information is available in the compiled code, however, and hence it is only applicable for statically-allocated data, of which the Linux tool `objdump` gives us the symbol mapping.

VHDL Tracing. Instead of having the VHDL simulation dump traces, and then verifying the correctness of them, we decided to let the VHDL code verify itself. To that extent, the VHDL implementation reads the FU traces from the cycle-accurate simulation, and then raises assertions when the values observed during the VHDL simulation differ from them. After having validated the cycle-accurate simulator, this on-the-fly trace comparison would allow us to debug, and eventually validate, the VHDL code. Unfortunately, the trace comparison is not without problems. Because these two simulators operate at different abstraction levels, defined behavior in one simulator can correspond to undefined behavior in another, in which cases false-positive assertions are raised.

As a workaround, the cycle-accurate simulator tags the values in a trace with tags that indicate whether a value is defined or not. The addition of the correct

tags was a time-consuming process, largely guided by trial and error. We feel this cannot be avoided however, as the goal of having an RTL-level cycle-accurate μ-arch simulator is precisely to enable the verification at a more abstract level.

Furthermore, when loops are mapped onto the array mode, many instructions are executed speculatively to shorten the critical code paths. This again implies that some operations are executed on undefined operands. To detect those during the verification of the tool chain, all possible operands are initialized to easily detectable values in the compiled and in the cycle-accurate μ-arch simulator. For example, the value `0xdeadcafe` is well suited because the human eye spots it easily in traces. In hardware, these values do not occur of course, and hence they were replaced by zeroes to generate traces for automated trace comparisons.

4 Conclusion

The ADRES CGRA was developed from a compiler perspective to ensure that C-code could be mapped onto it automatically. This paper presented some of the issues that were dealt with during the development of a hardware implementation and a concrete instruction set, and the verification of all (new) back-end tools and code. All issues were resolved in relatively simple ways, without imposing impractical or unrealistic constraints on either the source code or on the ISA.

Acknowledgement. This research has been carried out in the context of IMEC's multimode multimedia program which is partly sponsored by Samsung and Freescale Semiconductor.

References

1. August, D. I., Sias, J. W., Puiatti, J.-M., Mahlke, S. A., Connors, D. A., Crozier, K. M., and mei W. Hwu, W. The program decision logic approach to predicated execution. In *Proc. ISCA '99* (1999), pp. 208–219.
2. Berry, G. The foundations of esterel. *Proof, language, and interaction: essays in honour of Robin Milner* (2000), 425–454.
3. Hartenstein, R. A decade of reconfigurable computing: a visionary retrospective. In *Proc. of Design, Automation and Test in Europe (DATE)* (2001), pp. 642–649.
4. Mei, B. *A coarse-grained reconfigurable architecture template and its compilation techniques*. PhD thesis, Katholieke Unvirsiteit Leuven, 2005.
5. Mei, B., Vernalde, S., Verkest, D., Man, H. D., and Lauwereins, R. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *International Conference on Field Programmable Technology* (2002), pp. 166–173.
6. Mei, B., Vernalde, S., Verkest, D., Man, H. D., and Lauwereins, R. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field-Programmable Logic and Applications* (2003).