Integrating Custom Instruction Specifications into C Development Processes

Jack Whitham and Neil Audsley

Department of Computer Science, University of York, York, YO10 5DD, UK
`(jack|neil)@cs.york.ac.uk`

Abstract. We describe a new approach for creating hardware description language (HDL) specifications for custom instructions, to form part of the instruction-set architecture (ISA) of an application specific instruction set processor (ASIP). Our approach integrates fully into the traditional C development process, binding tightly with software source code and simplifying the ASIP optimisation process. Our tool is also free software, facilitating its use in future research.

1 Introduction

Increasing system efficiency by extending processors with application specific instructions has been considered widely, with many commercial products available [10, 5, 3]. However, existing commercial and research solutions separate the description of custom instructions from the actual software using these instructions. This paper proposes an integrated approach, allowing the user to specify custom instructions within the software source code itself.

This approach simplifies build processes by integrating into industry-standard compilation tools, permits easy testing of custom hardware, and makes full use of existing compiler optimisations. C programmers will need little specialist hardware design knowledge to make use of our approach. No new language needs to be learnt, the tool flow is the same as that used in most Unix `make` files, and separate compilation is fully supported.

Application-specific instruction set processor (ASIP) tools [10, 5] generate ASIP cores. [15] gives a good overview of ASIP design methodologies. Today, ASIPs are usually soft processor cores, intended for use as part of an FPGA design. Typical ASIP tools permit rapid optimisation of a hardware platform to a particular application. The processor can be tuned to developer requirements, such as reducing power consumption or increasing speed.

But the tools either do not attempt to integrate software and hardware development [8], or do so only within a graphical integrated development environment (IDE) [4, 2, 23], which constrains development options to those foreseen by the tool vendor. Additionally, the tools are closed, inextensible software, generally operating on secret ASIP cores. This reduces their utility to researchers, who are unable to adapt the tools for experimental purposes. Some adaptations of ASIP technology, such as the introduction of reconfigurable functional units

(RFUs) [26, 25], are currently impossible without either a manual implementation or a new tool.

In this paper, Sect. 2 examines the state of the art in ASIP tools and research. Section 3 discusses our approach and the implementation of our tool. Section 4 deals with our evaluation process and Sect. 5 concludes.

2 Existing ASIP Tools

2.1 General Overview

ASIP tools allow an existing "base" processor core to be customised to an application, by providing instruction set customisation (add or remove instructions), architectural customisation (add or remove execution, control or storage elements), and interface customisation (alteration of bus size and type).

Figure 1 illustrates a typical use of instruction set customisation. Software code is replaced by a custom instruction. Instruction fetches and clock cycles are saved. Overall, this approach may permit a smaller processor to be used, or slower, cheaper hardware may become usable. The cost, size and/or power consumption of the entire system may be reduced. Correct choice of code is essential [22, 7], but outside the scope of this paper. The typical approach involves profiling the application to find the most frequently executed code [4].

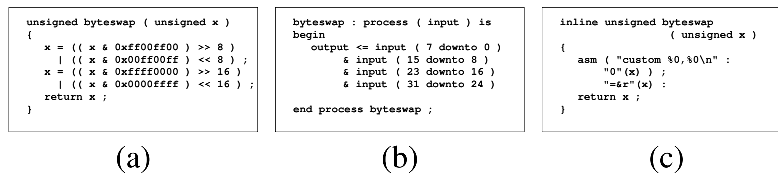

Fig. 1. Acceleration of C function (a) through dedicated hardware (b), accessed by a special opcode (c).

2.2 Commercial Tools

Tensilica Corporation [23] sells ASIP cores and tools under the trade name "Xtensa"[10]. The "Xplorer" IDE is used to modify the Xtensa cores. Custom instructions are described using the Tensilica Instruction Extension (TIE) [24, 7], a proprietary Verilog-like HDL. Architectural and interface customisation are also available: for example, multiplier execution units can be added if the application requires them. ARC Corporation [3] makes a similar set of tools for their own ASIP cores, including the "ARCitect" and "Metaware" IDEs, for building ASIP cores and software to execute on them.

ASIP Meister [5] provides an IDE for defining ASIP core features, but lacks features for software development. ASIP Meister permits a higher level of customisation than ARCitect and Xplorer - it is possible to change instruction encoding and processor microcode, even to the extent of implementing other processors [18].

Coware sells the LISATek tool [8], a general purpose processor definition tool based on the Language for Instruction Set Architectures (LISA). Coware's tools generate a compiler, simulator and VHDL processor model from a LISA description.

2.3 Development using an ASIP tool

Tensilica tools make use of the TIE [24] language for instruction specification. ASIP Meister allows custom instructions to be specified in a GUI. Other tools follow one of these two models: hardware is specified separately from software, and then used from the software in some way (e.g. compiler macros, in the case of Tensilica). This has the advantage that changes to the software are independent of the custom instructions. Changes to software alone will not force the ASIP to be rebuilt.

However, it has the disadvantage that the two descriptions are kept separate. This forces poor programming practice - functionally related items are in separate files, making the program harder to understand, debug, change and test. Programmers should aim to keep interfaces between modules to a minimum, but the TIE and ASIP Meister methodologies force an inter-module link for every custom instruction.

2.4 ASIP Research

ASIP technology predates the use of FPGAs. The term was first introduced to describe any processor designed for a particular application, not just a soft core to which instructions could be added.

ASIPs in today's form are a development of research into classic co-design, a methodology discussed in [13, 19]. Co-design tools are enhanced compilers that produce both a hardware description and a software binary for a particular application, with the intention of producing a faster implementation than software alone. This is done by migrating code fragments between hardware and software implementations: *partitioning* the program.

ASIP researchers initially attempted to derive the best ASIP instruction set for a program in its entirety [1], echoing the work of co-design researchers. Later work took a different direction, starting with a base instruction set and adding new instructions where necessary. This was more effective, as common instructions are always needed. Some ASIP tools try to automatically derive the partition [6, 22, 7, 11], but this is by no means essential. The automatic ASIP design problem has the same limitation as classic co-design: an exponential number of possible partitions.

All ASIP tools allow the developer to define the partition by hand, making use of the developer's understanding of the problem as a guide to partitioning. The developer may also directly define the hardware for each custom instruction. This avoids the suboptimal nature of automatic software hardware to translation [12].

2.5 ASIPs as a Basis for Research

Generally, existing ASIP tools are a good basis for research provided that they can be used as intended. Tensilica's Xtensa tools have formed the basis for some academic work, for example [22, 7]. [22] cites the flexibility of Xtensa as the reason for its choice. The work required both a processor with an extensible instruction set, and tools that provided easy access to the extensions. Xtensa provides both of these. Similarly, [21] chose LISATek over ASIP Meister as it provides direct access to the underlying processor definition language.

However, existing ASIP tools were not useful during the development of Chimaera [26], in which hard-wired custom instruction units are replaced by a run-time reconfigurable unit. No existing tools have support for such designs, and since existing tools are not open technology, they are not sufficiently extensible to act as a basis for fundamentally new designs. Thus, the Chimaera researchers were forced to start from scratch.

2.6 A Free ASIP: OpenRISC

The OpenRISC processor [16] is a freely available soft processor core, with a MIPS-like architecture and a five-stage single issue pipeline. OpenRISC already has ASIP features, but lacks ASIP tools.

Space is available within the instruction set for extensions to be added, and stubs exist within the Verilog source to allow custom execution units to be implemented. Additionally, some OpenRISC features can be "switched on" using definitions in a configuration file, allowing architectural customisation. OpenRISC also has a complete GCC tool chain, and similar performance to other 32-bit RISC soft cores.

3 Our Approach

3.1 Design Choices

Section 2.1 described the common features of ASIP tools. Of these, we consider instruction set customisation to be the most important, as it has a high potential for improvements that are easily quantifiable [22, 11]. It is this feature that our tool provides.

We have based our approach and tool on the C language and the free GNU C Compiler [9]. Despite its many shortcomings, C remains a widely-used language for embedded system development, and it is thus a good starting point. We chose not to define a new hardware description language, as Tensilica did with

TIE, instead making extensions to C and allowing hardware descriptions to be specified using a C subset. The subset is restricted: for example, at present, only single clock cycle operations are supported (see Sect. 3.4).

Noting that custom instructions are often only a small part of an application's software, and that functionally-related lines of code should be close together, our approach requires the programmer to specify hardware within software. Figures 2 and 3 illustrate our tool flow.

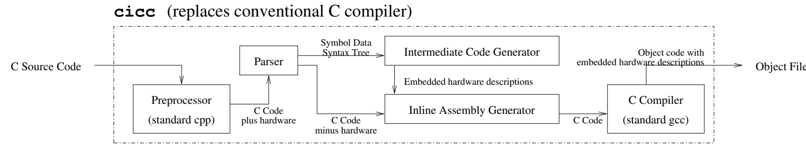

Fig. 2. Tool flow (a): The Custom Instruction C Compiler (`cicc`) is used in place of the regular C compiler. Note that both the hardware and software for each C module are placed in the same object file.

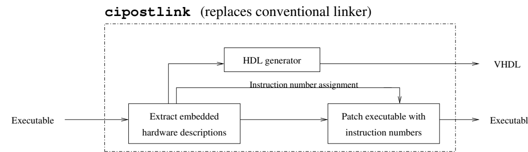

Fig. 3. Tool flow (b): After object files are linked in the usual way, the Custom Instruction Post-linker (`cipostlink`) extracts hardware descriptions from the executable, and then patches the executable to assign custom instruction numbers.

The traditional Unix `make` development process is used in place of an IDE. However, a generic IDE may be used. Our approach is fully modular, never requiring a compiler rebuild, and only forcing a processor rebuild if a custom instruction definition changes.

3.2 Data Paths

In a typical RISC architecture, only two input and one output bus are linked to an execution unit. The OpenRISC custom instruction unit is subject to this limitation, which is adequate for all RISC instructions, but not necessarily for custom instructions.

For example, an encryption operation will typically require both plain text and key inputs. Access to more than two registers is required. One solution to this problem is to provide access to all CPU registers. However, this requires register assignments to be fixed at the time of custom instruction generation, making

the register file non-orthogonal, as it now includes special purpose registers. In this environment, optimisations are less effective.

Local registers within the custom unit are a better solution. These are single-purpose registers, programmed with appropriate data as required. This approach does not break GCC optimisations, but it does force the programmer to consider thread safety, as the local registers cannot be saved during context switches. Despite this, it is a simple and effective solution.

3.3 Describing Custom Instructions

Suppose that we wish to write a custom instruction that replaces C code:

```
x = ( a + b + 4 ) & b ;
```

This would be written in Tensilica's TIE [24] language as:

```
operation Op ( out int x , in int a , in int b ) {
    assign x = ( a + b + 4 ) & b ;
}
```

The TIE code will be in a separate file, called from the main program using a generated `Op` macro: `x = Op ( a , b )`. The `Op` macro will expand to an appropriate instruction within the C file, in the form of inline assembly code. Meanwhile, the TIE code will be compiled separately into Verilog or VHDL, for inclusion in the basic ASIP core provided by Tensilica.

In our approach, the custom instruction would be described in the C source itself. The meaning of the operation is no longer hidden in a separate file. Here is the same example:

```
hardware {
    x = ( a + b + 4 ) & b ;
}
```

`hardware` marks the statement following it as a custom instruction. `hardware` statements are extracted automatically by our pre-compiler `cicc` (Fig. 2), which carries out the work of TIE. `cicc` is used in place of the regular C compiler (GCC): it is intended to be used as a drop-in replacement.

Pre-compilation is the process of extracting `hardware` statements and generating hardware descriptions and inline assembly for each. `cicc` does this by analysing the source file, building a complete symbol table, and then converting each `hardware` statement into a syntax tree, which is then converted into an intermediate hardware description language.

The pre-compilation step ends with the generation of inline assembly code, to call the generated hardware, and a new `.hardware` section within the generated object files, to contain the hardware description. This description may be compiled for simulation, or synthesised as hardware. Links are maintained between inline assembly and the associated hardware description using relocatable symbols. The post-linker, `cipostlink`, is run on the final executable to extract intermediate code and emit a description of the custom unit as VHDL (Fig. 3).

3.4 Custom Instruction Language

`hardware` blocks may contain a subset of C. The subset includes most expressions, but no loops. Loops cannot be permitted as all operations within a single `hardware` block must complete within a single clock cycle. Memory accesses are not permitted at present, as a modification to the OpenRISC processor would be required to support them.

The permitted statements are all arithmetic and logical expressions, except division and modulo, conditionals (`if/else` and `x?y:z`), assignments, and variable declarations and type casts. Accepted variable types are also restricted to `char` (8 bit), `short` (16 bit), `int` (32 bit) and `long long` (64 bit).

Variables may cross the interface between a `hardware` block and the surrounding C code, but at most two different external variables can be read from with the block, and at most one external variable can be written (see Data Paths, Sect. 3.2). This restriction is offset by local registers, declared using the `localregister` keyword, and temporary variables (declared in the local scope). These variables may be accessed without limitation.

4 Test and Evaluation Process

Evaluation of our free ASIP tool was carried out in two distinct areas: a test for correct operation, and a cost/benefit evaluation against other approaches.

4.1 Test Platform

To test correctness and efficiency, a variety of benchmark programs were set up to run on our target platform, a Xilinx Spartan-2E FPGA with 512Kb of 32-bit SRAM and a serial port attached. The platform is clocked at 12.5MHz. The FPGA holds an OpenRISC processor with a hardware multiplier, in addition to a boot ROM, drivers for the serial port and memory, timer, and hardware profiler. A Linux PC is able to download bitfiles to the FPGA via a parallel interface, then download software and obtain results via the serial port.

On the test platform, the OpenRISC version of the RTEMS [20] operating system is used as the host for the various benchmark programs. Although RTEMS adds a significant memory overhead (160Kb of code), it does not reduce execution speed when used in single-task mode. The Unix-like API of RTEMS makes it easy to compile the benchmark programs.

On the Linux PC, our `cicc` and `cipostlink` tools are installed in addition to the OpenRISC cross-compiler (GCC 3.2.3) and the Xilinx FPGA build tools (Xilinx ISE 7.1). Scripts were written to control the build process: each benchmark program can be built in “normal” mode, in which only standard OpenRISC instructions are used, or in “custom” mode, in which custom instructions are generated, built into the OpenRISC core, and then used.

The “normal” mode program is code from either the MiBench [14] or MediaBench [17] suite. We chose applications from these benchmark suites as representatives of the real applications that ASIPs are used in. Each benchmark

was minimally modified to run on RTEMS/OpenRISC, with no change in the test data used, and support for our hardware profiler and timer was added. These features allowed the time taken by the benchmark to be examined and improvements to be evaluated.

The “custom” mode program is almost the same as the “normal” mode program, but preprocessor directives (`#ifdef`, etc.) are used to substitute custom instructions for normal code in appropriate places. All other source is unchanged, and the same optimisation settings are used.

Our scripts are able to run an automated test cycle, in which benchmarks are built, downloaded onto the FPGA and tested. Table 1 lists the benchmarks used.

Table 1. The benchmarks that were used, and the relative efficiencies of their “normal” and “custom” implementations.

Benchmark name	Clock cycles, normal mode	Clock cycles, custom mode	Total hw (LUTs)	Extra hw (LUTs)	Max clock freq (MHz)	Speedup factor
basicmath	1456m	1449m	5958	1312	28.4	1.01
crc32	13m	11m	4840	194	30.9	1.23
dijkstra	772m	636m	4785	139	30.9	1.21
fft	192m	191m	5958	1312	28.4	1.01
g721	886m	446m	4672	26	30.9	1.99
jpeg	25m	19m	6069	1423	21.8	1.31
mad	129m	123m	5329	683	28.8	1.05

Having run each benchmark in “normal” mode, we examined the profile data from the built-in profiler to find the correct places to add custom instructions. The profiler identified a clear candidate in every case, such as:

– The `quan()` function in g721,
– the core of the CRC routine in crc32,
– 64-bit multiplier code in fft and basicmath, and
– fixed-point multiplier code in mad.

Each of these candidates was replaced by a small number of custom instructions, resulting in speed-ups at the cost of extra hardware, as shown in Table 1.

4.2 Operational Testing

Built-in Testing A simple extension to `cicc` provides testing support. As the syntax of each custom instruction is a subset of C, it is possible to automatically add a test harness during the pre-compilation step. The step compares the result of the operation carried out in hardware with the result from software. In the event of a mismatch, a failure function is called with information about the location of the error. This test approach was used for every example, which ensures that the generated hardware matches the original specification.

Checksum Testing All of the benchmarks produce some output. Checksums were used to ensure that the output of each benchmark did not change between the unmodified benchmark code, the "normal" mode benchmark, and the "custom" mode benchmark.

Functional Verification It is important that each feature that can be placed within a `hardware` block works correctly. Fortunately, due to the support for built-in testing, this is easily arranged. Across all the benchmarks, all the available features of the `hardware` block were used, and therefore tested.

4.3 Cost/Benefit Evaluation

Efficiency Table 1 indicates the improvement gained in each benchmark, plus the additional hardware cost in look-up tables (LUTs) and the change in maximum clock frequency. On our hardware, OpenRISC and associated hardware drivers run at up to 31.0 MHz, and take up 4646 LUTs. Each custom instruction will require some additional LUTs. The maximum clock frequency may also be affected by some custom instructions, if a new critical path is added. Our jpeg benchmark includes a custom instruction with a new critical path: its presence reduces the maximum clock frequency.

It is clear from this data that the tool can provide speed-ups for the various benchmarks. However, its efficiency in comparison to other ASIP tools is not obvious, and cannot be evaluated without access to those tools, which is not available for cost reasons. As comparisons with vendor-supplied benchmarks are only useful if all variables can be standardised, we decided to evaluate efficiency by comparison to the best possible case - direct manual implementation on hardware.

Manual implementation is far more laborious than any ASIP approach. There is no tool assistance: the developer must modify the processor directly. However, the developer may optimise the hardware directly to match the application requirements. There is no intermediate layer, as with our C subset, or Tensilica's TIE language. Greater efficiency is possible at the cost of developer time.

Table 2 illustrates the difference between a manual implementation and a tool-driven implementation for some of our test cases. The same interface into the processor was used for all implementations. As can be seen, the two implementations are very similar in each case, although manual implementations generally require less hardware.

Expressiveness of Our Language As our language is a subset of C, it is easily used by any C programmer. The concepts required to use it are simple enough that C programmers will become custom instruction designers with very little effort. In this respect, it improves upon Tensilica's TIE language, which is really a language for hardware engineers, being based upon the Verilog language.

However, TIE is more expressive than our language. Firstly, TIE allows for direct bit manipulation: every variable is an array of bits, as in Verilog. Our

Table 2. Comparison of manually implemented custom units and automatically generated ones.

Benchmark name	Extra hw using tool (LUTs)	Extra hw, by hand (LUTs)	Max clock freq using tool (MHz)	Max clock freq, by hand (MHz)
crc32	194	169	30.9	30.9
dijkstra	139	105	30.9	31.0
g721	26	23	30.9	31.0
jpeg	1423	1265	21.8	23.8
mad	683	644	28.8	31.5

language only permits bit manipulation indirectly through the standard C bit operators. It is our intention that the use of these operators will be optimised out and replaced with direct bit manipulation during synthesis, but we cannot guarantee that this will always happen.

Secondly, TIE allows instructions to take several clock cycles. This feature is not available in our language at present.

Improvements on Other ASIP Tools Our approach permits ASIP programs to be built in several stages. This is useful if ASIP instructions are required within the C library, operating system, or other supporting libraries. Conventional software engineering processes do not compile these parts together - rather, the operating system and libraries are built into a software development kit first, and the applications are added later. But other ASIP tools force them to be compiled together if customisations are made.

Our approach also permits recompilation of single code modules. Even modules that use ASIP features can be changed: a rebuild of the ASIP itself is only required if the ASIP features change. The approach tightly binds hardware descriptions with the code that uses them, making code more maintainable and well structured. Despite this, the separation between hardware and software is explicit and controlled by the programmer. Nothing is inferred or guessed "intelligently" by our system, so nothing can be guessed wrongly.

Although our approach only works for the OpenRISC processor at present, the interfaces could be adapted for most soft core processors for which HDL is available.

Our approach may appear to permit each custom instruction to be used once. However, this is not the case. Identical custom instructions are merged by the `cipostlink` program, permitting custom instructions to be replicated explicitly (by macros and inline functions) and implicitly (by GCC optimisations).

Disadvantages of Our Tool We do not consider the additional pre-compilation and post-linking steps to be a disadvantage, as they integrate into traditional `make` files and take very little time.

However, a serious disadvantage of our tool is the single clock cycle limitation. Loops must involve several instructions, and complex operations cannot be pipelined within a single instruction. The Tensilica ASIP tools are not subject to this limitation, so more complex custom instructions can be written in the TIE language without adverse effects on the clock frequency.

The tool also limits the data types that are usable from a custom instruction. Floating point is not available, and nor is integer division.

The tool does not yet try to merge hardware within the custom unit, when it can be shared between two or more instructions. This results in some instructions requiring far more hardware than strictly necessary. We rely on the synthesis tool to optimise the custom unit, but that tool does not have access to all of the available information about the function of each instruction. Thus, the tool could do a better job of optimisation with an instruction merging extension.

5 Conclusion

We have described a new approach for the generation of ASIPs, in which hardware descriptions for ASIP custom units are specified within the software code that makes use of them. Our approach is intended to integrate well into traditional development tool flows, permitting separate compilation and acting as a plug-in replacement for GCC. This allows ASIP features to be used within large projects.

We have also demonstrated our approach using a prototype tool, and showed its effectiveness using a series of benchmarks. Topics for future work will include the implementation of an optimiser for the custom unit inside `cipostlink`, the possible addition of code to permit multi-cycle instructions, and support for RFUs.

References

1. A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An ASIP instruction set optimization algorithm with functional module sharing constraint. In *Proc. ICCAD*, pages 526–532, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
2. Anonymous. Nios II Custom Instruction User Guide. Manual UG-N2CSTNST-1.2, Altera Corporation, 2004.
3. ARC International. Home page (accessed 16 Jan 06). http://www.arc.com/.
4. ARC International. Integrated profiler (accessed 16 Jan 06). http://www.arc.com/-software/developmenttools/codeprofiling/integratedprofiler.html.
5. ASIP Meister. Home page (accessed 16 Jan 06). http://www.eda-meister.org/asip-meister/.
6. N. N. Binh, M. Imai, A. Shiomi, and N. Hikichi. A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts. In *Proc. DAC*, pages 527–532, New York, NY, USA, 1996. ACM Press.
7. N. Cheung, J. Henkel, and S. Parameswaran. Rapid configuration and instruction selection for an ASIP: a case study. In *Proc. DATE*, page 10802, Washington, DC, USA, 2003. IEEE Computer Society.

8. Coware Corporation. LisaTEK Datasheet (accessed 16 jan 06). http://www.coware.com/PDF/products/LISATek.pdf.
9. Free Software Foundation. GNU Compiler Collection (accessed 16 Jan 06). http://gcc.gnu.org/.
10. R. E. Gonzalez. Xtensa — A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.
11. D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proc. CASES*, pages 137–147, New York, NY, USA, 2003. ACM Press.
12. B. Grattan, G. Stitt, and F. Vahid. Codesign-extended applications. In *Proc. 10th Int. Symp. Hardware/Software Codesign*, pages 1–6, 2002.
13. R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993.
14. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. 4th IEEE Workshop on Workload Characterization*, 2001.
15. M. K. Jain, M. Balakrishnan, and A. Kumar. ASIP Design Methodologies: Survey and issues. In *Proc. VLSID*, page 76, Washington, DC, USA, 2001. IEEE Computer Society.
16. D. Lampret. OpenRISC 1200 (accessed 16 Jan 06). http://www.opencores.org/.
17. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Int. Symp. Microarchitecture*, pages 330–335, 1997.
18. Masaharu Imai. ASIP Meister - DAC participation information (accessed 16 jan 06). http://www.sigda.org/programs/UniversityBooth/Ubooth2002/p/infodetail.html.
19. G. D. Micheli, W. Wolf, and R. Ernst. *Readings in Hardware/Software Co-Design*. Morgan Kaufmann Publishers Inc., 2001.
20. OAR Corporation. RTEMS (accessed 16 Jan 06). http://www.rtems.com/.
21. Scharwaechter, H. and Kammler, D. and Wieferink, A. and Hohenauer, M. and Zeng, J. and Karuri, K. and Leupers, R. and Ascheid, G. and Meyr. H. ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study. In *Proc. SCOPES*, Amsterdam (Netherlands), Sept 2004.
22. F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. Synthesis of custom processors based on extensible platforms. In *Proc. ICCAD*, pages 641–648, 2002.
23. Tensilica Corporation. Home page. http://www.tensilica.com/.
24. Tensilica Corporation. TIE: Product brief (accessed 16 Jan 06). http://www.tensilica.com/pdf/TIE_T1050.qxd.pdf.
25. D. Vassiliadis, N. Kavvadias, G. Theodoridis, and S. Nikolaidis. A RISC architecture extended by an efficient tightly coupled reconfigurable unit. In *Proc. ARC*, 2005.
26. Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. 27th Int. Symp. Computer Architecture*, pages 225–235, 2000.