An ISA-Retargetable Framework for Embedded Software Analysis

Gabriele Luculli
STMicroelectronics
Advanced System Technology Group
38000 Grenoble, France
gabriele.luculli@st.com

Abstract

Industry requires new advanced tools and methodologies for the design of complex system-on-chip platforms. In STMicroelectronics we developed an innovative retargetable technology for the analysis and optimization of embedded software. In this paper we provide an overview of our retargetable tool chain which consists of: a specification reader, a disassembler, a code analyzer and a functional simulator. Their capabilities, specific features and limitations are described in detail.

1 Introduction

Since the invention of microprocessor, instruction set simulators have been the main tools used by engineers to design, analyze and optimize the software. As microprocessors have evolved in complexity of the instruction set, the number of integrated transistors and innovative microarchitecture features, the related simulation technology has advanced as well in order to improve the simulation performance and, more in general, to provide better answers to such evolving needs. The initial focus on functional simulation was very soon shifted toward the performance simulation, due to the introduction of pipelining in the microprocessor architecture. Since then, the efficient modeling of new microprocessor features has been the main theme on each new generation of simulators, and the simulation performance has been the dominant metric to establish the simulator quality.

Today, the advent of system-on-chip (Soc) platforms and the steadily growing presence of embedded software ask for a new simulation technology. Like the old days we still need simulators to validate any software design but, in addition, new design activities require the use of some sort of simulation technology which is able to provide additional quantitative information on the embedded software. The reuse of software libraries for different products is probably the largest single challenge facing a software engineer. The porting of an algorithm to a particular platform is the need to optimize the algorithm for that platform. This means to analyze first, and eventually to rewrite then, the software in order to exploit the available processor microachitecture, the memory architecture or the available hardware accelerators. Performance optimization is the most common criteria but, with the advent of mobile computing, energy consumption is assuming a new leading role as well [11]. It is also interesting to observe what happens in the today's practice when we reverse the perspective, looking at software by the hardware side rather than at hardware by the software side. We can easily realize that microprocessor architects face exactly the dual challenge of software engineers. Even if the evolution of microprocessor architecture still critically depends upon the advancement of the technology, such as scaling of the feature size and improvements in the lithography process, program behaviour has become increasingly important in defining architecture tradeoffs. The usefulness of a complex processor implementation is today severely limited by the unpredictable events occurring during the software execution [1], so that it becomes essential for the low-cost embedded markets to develop architectures which are specific for each application domain [4]. Simplifying the hardware and placing the responsability of achieving the design optimization on the software becomes a reasonable alternative. Whether we consider the software or hardware side, the need of a better understanding of the program behaviour and its interaction with the hardware architecture are became an important priority. We think that simulation technology may have a major role if it will be able to provide the required flexibilities without sacrificing the simulation performance.

In this paper, we provide an overview of an advanced retargetable technology for instruction-set simulators (ISS). This is part of a general environment for the retargetability of software tool chains which we developed in STMicroelectronics. The overall flow is driven by a formal specification of the instruction-set architecture (ISA) written in the

SLED language [10]. The reported results are innovative at least for:

- The successful development of a retargetable simulation technology for embedded software which is based on the correct-by-construction paradigm. To the best of our knowledge, this is the first time that a formal specification is exploited for this purpose.
- The introduction of a general selective back-annotation mechanism in the retargetable simulator in order to provide quantitative feedback of any hardware metric.
- The introduction of the code analyzer as new engineering tool to visualize the distribution of hardware metrics into the program structure.
- The reaching of the high simulation performance which is typical of compiled-simulation approaches but by using an interpretive technology.

Following, we summarize the most recent research results on retargetable ISS technology. After a short description of a case study, our retargetable tool chain is introduced and described with respect to the implemented functionalities. Conclusions and future work on tool retargetability complete the paper.

2 Related Work

In the past years, several approaches on simulators retargetability have been explored. Most of them share the same basic idea of exploiting some forms of processor description to generate automatically a simulator. However, they are usually based on non-formal specification languages so resulting on a quite week form of ISA retargetability.

The method proposed by Pees et al. [9] is probably one of the most interesting. They use the machine description language LISA to generate an almost complete tool chain consisting of assembler, linker, debugger and simulator. In particular, they describe their technology for compiled simulation providing wide evidence of the state-of-art performance of the simulators they generated. Unfortunately, the LISA language just provides an almost exhaustive representation of the instruction set. Neither the structure of the ISA encoding, nor any form of instruction semantic can be described in a completely formal way. So, there is no way to validate correctness, completeness or even non-ambiguity of a processor spec. Since the input spec is not validated to be correct, any generated tool, included the simulator, could eventually be wrong implemented.

A quite similar approach is described in [3], even if they just focus on simulation technology. They make use of a machine description which specifies the encoding and semantics of each instruction in the instruction set, as well as some global data structures and action descriptors of the simulator. Since their main aim is a simple implementation of the simulator generator, they adopt an almost straightforward C-like syntax and semantic. The main claimed benefit is that they don't have to parse and analyze any hardware description language, so that most of the generated C code for the simulator is just code which is directly pasted in from the machine description. Nevertheless, the same drawbacks we previously remarked apply to this approach as well. Moreover, the effort required to develop a new simulator seems roughly equivalent to a full custom approach.

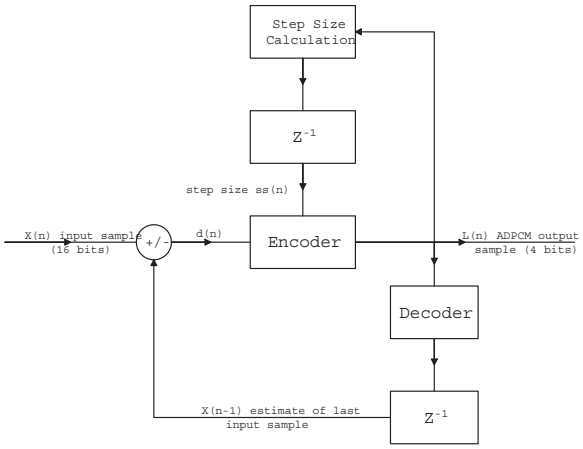

Figure 1. Block diagram of the ADPCM encoding algorithm

A complete retargetable tool chain was developed by Hartoog et al. [2] using the nML machine description language. It is interesting that they can derive both an interpreted and a compiled simulator from the same ISA description, so their approach seems to be very flexible. Unfortunately, such flexibility is paid in terms of simulation speed which is generally quite low, i.e. 26 KIPS and 150 KIPS for an ARM7 target on a SPARC20 host machine respectively for interpreted and compiled simulation. In [5] the authors pursue similar goals starting from an HDL description of the target processor and achieving equivalent simulation speeds.

3 The ADPCM Application

Through the whole paper we use a very simple embedded application, the ADPCM application, as main case study in order to practically expose the utilization of our retargetable simulator and, more in general, our tool chain. Some details on its main characteristics are here provided for completeness.

The Adaptive Differential Pulse Code Modulation (AD-

PCM) is a differential coding schema in which each sample approximates the difference between the present input value and the previous one. The weighting of the magnitude portion of the difference is adaptive. That is, it can change after each sample. Our implementation, which is quite common, takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1.

A block diagram of the ADPCM encoding schema is reported in Figure 1. A linear input sample is compared to the previous estimate of that input. The difference along with the present step size are presented to the encoder logic which, as outcome, produces the ADPCM output sample. This output sample is also used to update the step size calculation and to compute the linear estimation of the input sample. The following is a representation of the encoding logic in pseudocode:

```
let B3 = B2 = B1 = B0 = 0
if (d(n) < 0)
   then B3 =1
d(n) = ABS(d(n))
if (f(n) >= ss(n))
   then B2 = 1 and d(n) = d(n) - ss(n)
if (f(n) >= ss(n)/2)
   then B1 = 1 and d(n) = d(n) - ss(n)/2
if (f(n) >= ss(n)/4)
   then B0 = 1

L(n) = 10002*B3 + 1002*B2 + 102*B1 + B0
```

It is useful to observe that the code we used in our experiments differs a bit from this pseudocode. In fact, we optimized its implementation by replacing the division operations, which are really time expensive, in term of shift and add operations, which are very efficiently implemented in the ST200 microprocessor we used as reference.

4 Retargetable Tool Chain

The overall software architecture of our retargetable framework is depicted in Figure 2. It consists of three layers of software which incrementally add up to provide different views of the same basic technology to different users. In the lowest layer we implemented the building blocks which are common to one or more tools of the tool chain. A description of such modules can be found in [8] where, in addition, a complete analysis on the optimization of the Sim Library is reported. In the second one, by composing several building blocks of the previous layer, we provide the real *ISA-retargetable tool chain* which consists of a specification reader, a disassembler, a code analyzer and a functional simulator. In the last one we implemented a *software generator* which is able to generate automatically a *custom tool chain* for the given ISA spec. Obviously, the main difference between the last two layers

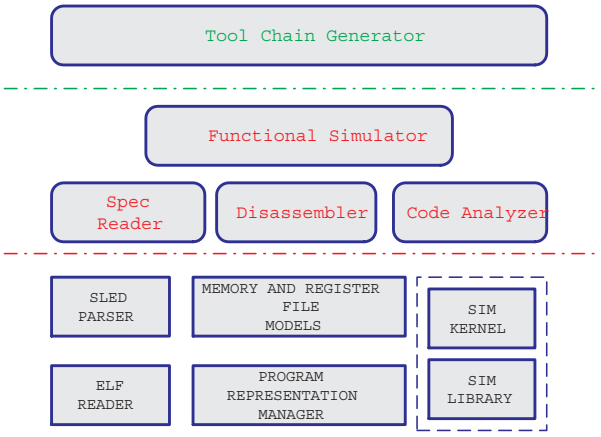

Figure 2. Global software architecture of our ISA-retargetable environment

is that, whether the processor description is used by any retargetable tool to adapt its functionality at run-time to the specified processor, the software generator uses the same description to generate a version of the tool chain (i.e. a *retargeted tool chain*) which is customized for the specific processor and, eventually, customizable for additional requirements.

4.1 The specification reader

Let's start to see in more details the ISA-retargetable tool chain. The first tool, named *spec reader*, is a general parser of the input SLED spec. Its main purposes are to validate the correctness of the specified encoding and to build an efficient internal representation of the recognized instruction set. In practice, this is the key tool for the correct development of any new spec. It is worth noting here the very special importance that is covered by the spec correctness in any correct-by-construction approach. The main benefits of this paradigm come from the well known fact that, if the input spec is correct, all the outcomes of the automation flow are guarantee to be correct. So, however complex are the final outputs, they don't need any further validation step. The clear advantage is the extreme reduction of the development time. Several details on the adopted internal representation and the decoding process can be found in [6].

4.2 The retargetable disassembler

Following there is the *retargetable disassembler*. Its basic functionality is to recognize the instructions of the input binary file and to print them in output using the correct assembler syntax. The main difference between our

retargetable disassembler and a classic disassembler is that the encoding format of the instruction set and the assembler syntax are fixed for the second one whereas they are variable for the first one. We adopted a SLED formal description of the ISA encoding in order to grasp the meaning of such variability. It is interesting to observe that, following the recognition of an instruction opcode, there are still several other aspects that have to be considered in a generic disassembler:

- The recovery of the instruction operands. Depending on the kind of encoding a single operand may be splitted across several different fields, and such fields distributed non-contiguously across the input bit stream.
- The correct signed/unsigned interpretation of the operand values. The same encoding field may be interpreted in different ways for different opcodes.
- The value of some operands may not be simply encoded in the input bit stream, but it may result from the implicit evaluation of some kind of formula on the fields they consist of. So a static pre-processing in term of elementary operations, such as shift and masking, is typically required to obtain the real value of the operand.
- Sometime the value of an operand is simply unknown in the binary code, e.g. the target address for a relative jump which is implemented by an absolute jump. Nevertheless, it must be resolved at decoding time in some way. Some kind of dynamic formula (i.e. a formula which makes use of a dynamic variable, like the program counter) is required in this case.

Fortunately, all such aspects were already taken into account by the authors of the SLED language in the adoption of its specific constructs. So, we have just exploited the available language features in order to provide the ISA retargetability of the disassembler. Following, an example of execution of our retargetable disassembler is here reported just to provide some hints on the previously quoted items:

```
--------------------------------------------------------------------------
Retargetable Disassembler
STMicroelectronics, AST Grenoble
(C) DSE Team

DSE Framework version 0.1.2 - Package ver. 0.1.2
compiled Wed Oct 16 17:36:26 MET DST 2002 by lucullig on gnx1190
Library libbasedse_nrm ver. 0.1.2
Library libbits_nrm ver. 0.1.1
Library libbase_utils ver. 0.1.1
--------------------------------------------------------------------------
File 'examples/Lxemb/adpcm_speed_o.elf'
Processor ST200 (Big Endian)
Entry Point: 0x08000000

Section n.  0 --> Initial Address 0x08000000 - Length: 0x00000018 (code)
Section n.  1 --> Initial Address 0x00000000 - Length: 0x00000000 (data)
Section n.  2 --> Initial Address 0x08000500 - Length: 0x00000340 (code)
Section n.  3 --> Initial Address 0x08000840 - Length: 0x000013CC (data)
Section n.  4 --> Initial Address 0x00000000 - Length: 0x00000000 (data)

08000000 [1584000EC8010300]:  mov $r12 = 08001C10 ;;
08000008 [15800080C800030C]:  add $r12 = $r12, 00010000 ;;
08000010 [F00001E0]:  call $r63 = 08000790 ;;
08000014 [F1000000]:  halt ;;

08000500 [081E030C]:  add $r12 = $r12, -020
08000504 [230023D3]:  ldb $r15 = 002 [$r19]
08000508 [C8001580]:  mov $r22 = 001 ;;
0800050C [21000553]:  ldh $r21 = 000 [$r19]
08000510 [00014480]:  mov $r20 = $r18
08000514 [0001E440]:  mov $r30 = $r17
08000518 [C000E400]:  mov $r14 = $r16 ;;
0800051C [C001F4C0]:  mov $r31 = $r19 ;;
08000520 [1584000808C8048F]:  sh2add $r18 = $r15, 08001080
08000528 [C00133C0]:  mov $r19 = $r15 ;;
0800052C [20000452]:  ldw $r17 = 000 [$r18]
08000530 [C0010540]:  mov $r16 = $r21 ;;
08000534 [C0000000]:  nop ;;
[...]
0800080C [C800138E]:  add $r14 = $r14, 001 ;;
08000810 [317FFFE4]:  goto 080007A0
08000814 [25000380D5040009]:  stw 08001200 [$r0] = $r14 ;;
0800081C [20000400D5040009]:  ldw $r16 = 08001200 [$r0] ;;
08000824 [E0010FCC]:  ldw $r63 = 010 [$r12] ;;
08000828 [C0000000]:  nop ;;
0800082C [C0000000]:  nop ;;
08000830 [C0000000]:  nop ;;
08000834 [31800000]:  return $r63
08000838 [C802030C]:  add $r12 = $r12, 020 ;;
0800083C [00000000]:  nop
```

4.3 The retargetable code analyzer

The third tool is the *code analyzer*. It has, as main functionality, the recovery of the program structure in term of global control-flow graph (CFG), local control-flow graph to each procedure and call graph. Classic tool chains don't usually include such kind of code analyzer because it is not useful to design general-purpose software: code implementation and functional verification are simply done by a try-and-fix approach using just a debugger. The situation is quite different for embedded software which requires a substantial performance tuning for each specific hardware platform. In fact following the simulation, software engineers make use of the code analyzer to identify the performance bottlenecks into the software structure so that they can hopefully re-write the original source code to enhance its performance. In the next section we will provide additional details on the integrated use of the code analyzer in the simulator.

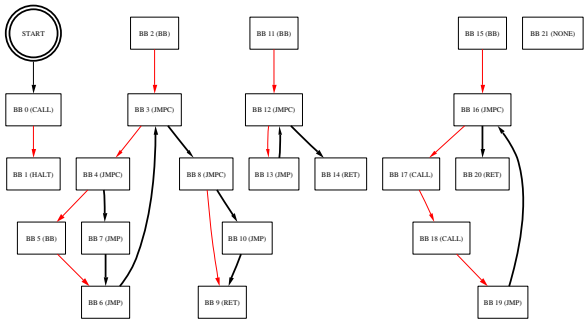

Image: adpcm_speed_o.elf
22 Basic Blocks, 22 Links

Figure 3. Global control-flow graph of the ADPCM application

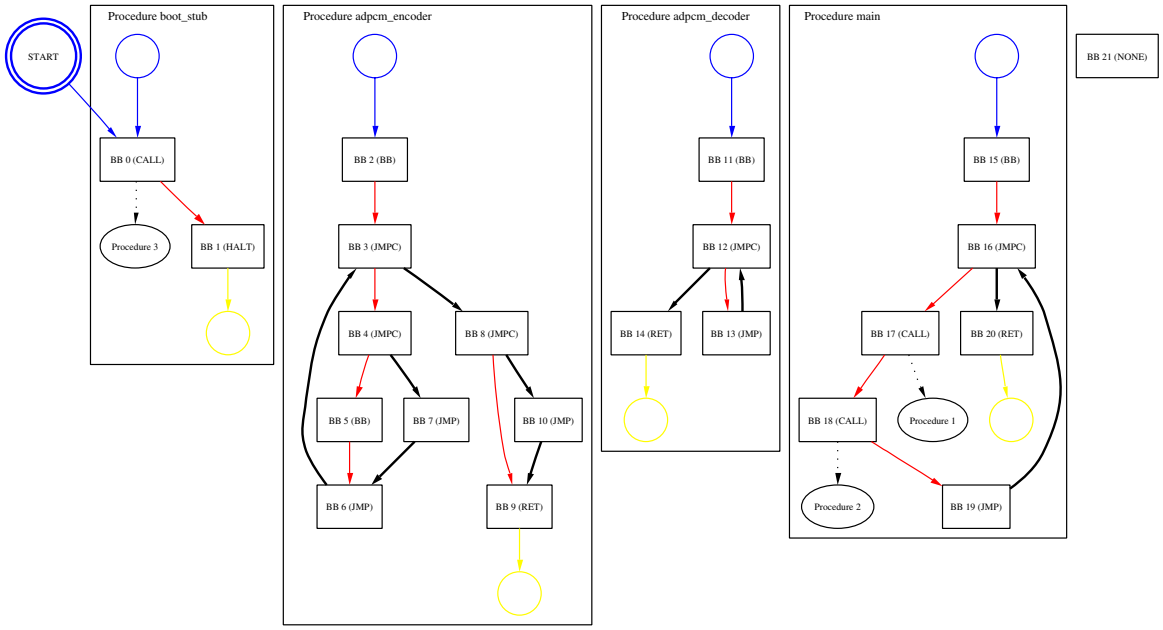

Image: adpcm_speed_o.elf
4 Procedures, 22 Basic Blocks, 30 Links

Figure 4. Decomposed control-flow graph of the ADPCM application

An example of recovered global CFG is reported in Figure 3. The identification of the basic block boundaries is done by a straightforward generalization of the classical algorithm implemented in any compiler [7]. The instructions which change the flow of execution are marked as end of the actual basic block, and their eventual target addresses are used to identify the starting boundaries of new basic blocks. The mentioned generalization just refers to how to identify the instructions to mark. For this purpose, we extended the SLED spec with a specific construct which is used to classify each recognized instruction [6]. As it is apparent in Figure 3, a classification of each basic block follows as well.

Starting from the global CFG, a second algorithm is used to recover both the boundaries of each procedure (Figure 4) and their call graph. This is mainly done by a couple of graph visits: a forward traversing which starts from the target address of each basic block marked as CALL, and a backward traversing which starts from each basic block marked as RET (i.e. its last instruction is a return instruction). Even if the reported example is not the case, it is quite frequent to have procedures with several entry points and/or several exit points. This is mainly due in embedded code to the extensive optimizations performed by the compiler. For this reason, we implemented a simple algorithm which starts, first, by discovering the basic blocks which are shared among several procedures. Then, it joins all the connected basic blocks into a single procedure with a common virtual entry node and/or a common virtual exit node. In this way, all the procedures are internally represented by a standard single-input single-output control-flow graph.

In summary, the retargetable code analyzer has the capability of recovering any static information which is related to the binary file. For the moment, we just focused on the CFG structure but, if it should be useful in the future, any other algorithm of static analysis could be easily adopted to recover additional information.

4.4 The retargetable functional simulator

The *retargetable simulator* is the most complex tool of the tool chain. It includes all the functionalities described in the previous sections, so the spec reader and the code analyzer are parts of him. Moreover, it provides the general technology for the functional simulation of any ISA. The main characteristic of our simulation technology are:

- It adopts a mixed interpretive-compiled simulation approach. Instructions are decoded just one time, as in compiled simulation, but then they are executed like

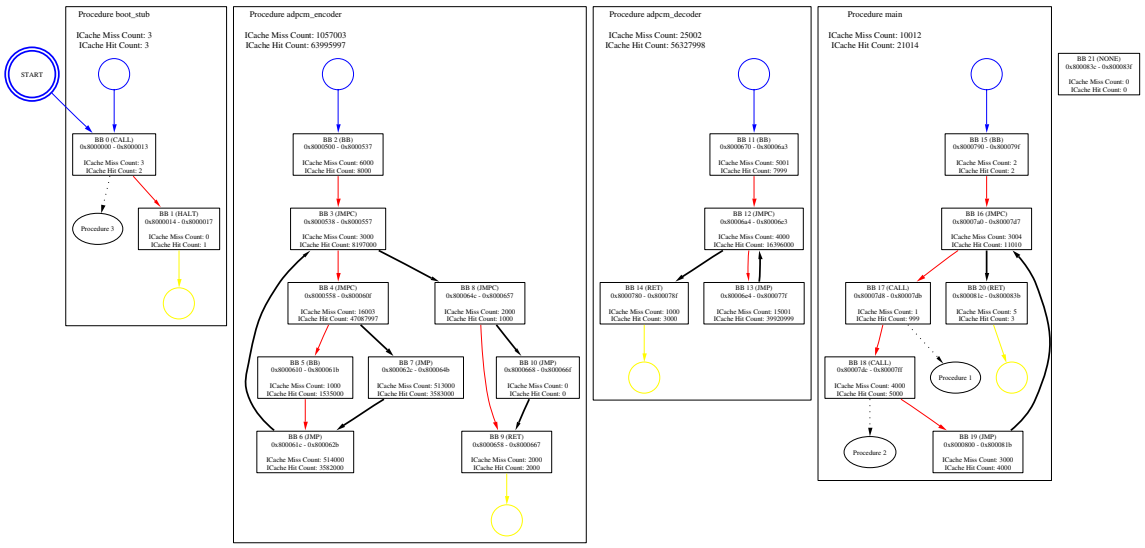

Image: adpcm_speed_o.elf
4 Procedures, 22 Basic Blocks, 30 Links

ICache: 256-byte size, 8-byte line
ICache Miss Count: 1092020
ICache Hit Count: 120345012

Figure 5. Backannotation of the ICache performance metrics into the program structure

in interpretive simulation (i.e. there is a single behavioral code for each instruction class). The spectrum of mixed simulation technologies is quite large. We feel we reached the boundary which provides almost the same high performance of compiled simulation, but without the typical drawbacks of code size explosion and expensive compilation time which characterize such technology.

- The adopted internal representation of the instruction set for simulation purpose is quite space consuming. In fact, all the instruction operands are statically allocated inside the representation of each instruction class, and then they are initialized at decoding time for each specific instruction of the binary file. This choice was mainly driven by our wish of discarding from the simulation critical loop all the timing consuming computations which not directly contribute to the simulation itself. There is a clear tradeoff between the size of the internal representation and the simulation speed. The practice shows that it is without any doubt convenient to gain simulation speed, even if some additional memory space is wasted. This is even more true when simulators are used to perform extensive design-space explorations as it is required for system-on-chip designs.

- The skeleton of the simulator (i.e. without instructions behavioral code) for a specific processor can be automatically derived from a typical SLED spec without the need to use any further information.

- The main simulation loop supports both CISC-RISC and VLIW simulation semantics. We successfully implemented the full functional simulators of both the ARM7 processor, a typical RISC machine, and the ST200 processor, an advanced VLIW machine. A partial simulator was generated for a reduced instruction set of the ST10 processor, that has most of the typical addressing modes of a CISC machine.

- It supports the hardwired modeling of hardware modules, to be attached to the core processor, by a general concept of module encapsulation and I/O ports which are implemented by ad-hoc C++ classes. The modules we already implemented are: the ARM7, ST200 and ST10 core; a multi-banks memory with variable size and word size; a direct-mapped cache memory for instructions, with variable cache and line size; a

Benchmark	Description	Size (bytes)	Simulation Speed
ac3enc	AC3 audio encoder	67.7K	1.13 MIPS
adpcm	Adaptive pulse code modulation encoder/decoder	832	1.25 MIPS
des	Data Encryption Standard	5.3K	1.05 MIPS
djpeg	Image decoder JPEG	34.5K	809 KIPS
fft	Cooley-Tukey Fast Fourier Transform 1024 points	52.8K	884 KIPS
matcnt	Runs a multiplication test on arrays	3.6K	1.05 MIPS
motion	MPEG2 Motion estimation algorithm	4.6K	1.08 MIPS
sort	Bubble sort algorithm	1.8K	1.31 MIPS
whetstone	Whetstone benchmark in C	43.9K	804 KIPS

Table 1. Benchmarks description (VLIW ST200 target)

direct-mapped write-through cache for data, with variable cache and line size.

- It includes a very general mechanisms to collect execution statistics for any modeled hardware module. First, one or more metrics (not necessarily functional metrics) can be defined for each given module. Then, a selective backannotation mechanism is capable of decomposing the collected statistics on the recovered code structure.
- The experimental results show a remarkable simulation performance. For the benchmark set of Table 1 the average simulation speed is within the range of one MIPS for a 750Mhz 1024 Mbytes Sun workstation.

Overall, our functional simulator provides many interesting flexibilities. The automatic ISA retargetability is most prominent but, we think, the available mechanism for metrics backannotation is without any doubt the most useful to quantify the hardware effects. A simple example of the possible simulation output is reported in Figure 5. It refers to the execution of the ADPCM application on a very simple architecture which consists of an ST200 core processor with a direct-mapped instruction cache and a main flat memory. Data are accessed directly in the main memory, whereas instructions are fetched from the cache. We have chosen an unrealistic cache size of 256 bytes just to produce some non-trivial effects for this small 832-byte application. Anyway, everything still works the same for any other hardware configuration. For the cache we have defined two classical metrics: the number of misses and the number of hits. These are very simple functional metrics but any other kind of metric, e.g. cache power consumption or average access time, could be defined as well, provided that they have been considered inside the hardware model.

Our backannotation mechanism is very general, yet very efficient. We support the definition of new metrics for a given hardware module by a specific C++ class which is used to formalize both the internal state and the semantic of such metrics. Any kind of metric can be automatically used by the backannotation mechanism to annotate the code structure. Moreover, the gathering of statistics is made effective by the possibility of choosing in a very selective way which parts of the code has to be annotated with which metrics. The two choices, part of code and metric, are completely orthogonal. So, simulation time is just consumed for the collecting of the demanded statistics which are relative to the demanded portion of code.

As example, let's consider Figure 5 which reports the annotation of the overall ADPCM application at both procedure and basic block levels with the previously cited cache metrics. In this case, the main goal of the analysis is to identify the cache bottlenecks so that the code can be re-written in an hopefully improved way. It is extremely easy to identify the `adpcm_encoder` procedure as the single prominent source of cache misses and, more specifically, the basic blocks BB6 and BB7 as source of the cache conflicts. Obviously, this is just a toy example without any practical implication. Anyway, the same approach applies in general to applications of any complexity and for whatever metrics of whatever hardware module.

5 Conclusions

We have presented an overview of our retargetable tool chain with a special emphasis on the retargetable simulator. A single formal specification of the instruction-set architecture is exploited to derive a correct-by-construction implementation. Disassembler, code analyzer and instruction-set simulator have been successfully generated for three different embedded processors (ARM7, ST200, ST10) resulting on functional and performance effective implementations. Several available flexibilities and features have been exposed with the help of simple examples.

Our future research includes the automatic generation and optimization of the behavioral code for the simulator starting from an extension of the actual encoding spec. This will be another major step toward the complete retargetability of simulation technology, as well as a great opportunity to improve further the simulation speed.

References

[1] M. J. Flynn. Basic issues in microprocessor architecture. *Journal of Systems Architecture*, pages 939–948,

1999.

[2] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proceedings of the 34th Conference on Design Automation (DAC-97)*, pages 303–306, NY, June 9–13 1997. ACM Press.

[3] T. E. Jeremiassen. Sleipnir – an instruction-level simulator generator. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, 2000.

[4] C. E. Kozyrakis and D. A. Patterson. A new direction for computer architecture research. *IEEE Computer*, 31(11):24–32, 1998.

[5] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators. In *Proceedings of the 1999 IEEE ASP-DAC*. IEEE, 1999.

[6] G. Luculli. Retargetable tools for embedded software. *Submitted to conference*, 2003.

[7] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[8] F. Papariello and G. Luculli. Optimization of a retargetable functional simulator for embedded processors. *IEEE Proc. of the 9th Annual International Conference and Workshop on the Engineering of Computer-Based Systems*, April 2002.

[9] S. Pees, A. Hoffmann, and H. Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Transactions on Design Automation of Electronic Systems.*, 5(4):815–834, Jan. 2000.

[10] N. Ramsey and M. F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[11] A. Sinha and A. Chandrakasan. Energy aware software. In *IEEE Proceedings of the 13th International Conference on VLSI Design*, 2000.