A compilation-based software estimation scheme for hardware/software co-simulation

A Compilation-based Software Estimation Scheme for Hardware/Software Co-Simulation

Marcello Lajolo
Politecnico di Torino
Torino, Italy
lajolo@polito.it

Mihai Lazarescu
Politecnico di Torino
Torino, Italy
lazarescu@polito.it

Alberto Sangiovanni-Vincentelli
University of California at Berkeley
Berkeley, CA, USA
alberto@eecs.berkeley.edu

Abstract

High-level cost and performance estimation, coupled with a fast hardware/software co-simulation framework, is a key enabler to a fast embedded system design cycle. Unfortunately, the problem of deriving such estimates without a detailed implementation available is very difficult.

In this paper we focus on embedded software performance estimation. Current approaches use either behavioral simulation with (often manual) timing annotations, or a clock cycle-accurate model of instruction execution (e.g., an instruction set simulator). The former provides greater flexibility (no need to perform a detailed design) and high simulation speed, but cannot easily consider effects such as compiler optimization and processor architecture. The latter provides high accuracy, but requires a more detailed implementation model, and is much slower in general. We hence developed a hybrid approach, that incorporates some aspects of both. It provides a flexible and fast simulation platform, considering also compilation issues and processor features.

The key idea is to use the GNU-C compiler (GCC) to generate "assembler-level" C code. This code can be annotated with timing information, and used as a very precise, yet fast, software simulation model.

We report some experimental results that show the effectiveness of our approach, and we propose some future improvements.

Keywords: compilation, software estimation, delay modeling.

1 Introduction

The design of a complete hardware/software system is becoming more and more complex, due to the progress made in various areas of hardware and software technology. In particular with the ability to mix processors, complex peripherals, and custom hardware on a single chip, it is unthinkable to address full-system design and analysis with a manual approach. This complexity demands a new methodology and set of tools.

Co-simulation at an early design stage, using a *predictive* performance model plays a key role in a complete system-level design environment. It can evaluate the feasibility of a particular hardware/software partition, processor choice and so on, earlier than by using the traditional methodology, based on separate implementation paths for hardware and software.

However, this high level performance estimation of hardware/software real-time embedded systems is very difficult. On the software side (the focus of this paper), the main problem is the prediction of the performance of a program written in a high-level language (in this case C) on a given processor architecture. It must take into account several architectural effects, such as compiler optimization, memory hierarchy (registers, caches, ...), pipelines, multiple functional units, just to name few of them.

The approach presented in this paper is aimed at filling the performance estimation gap in the POLIS [1] embedded system design environment. POLIS uses two different software estimation schemes, that are representative of much broader classes of techniques:

1. a high-level estimation methodology, based on source code analysis (software delay macromodeling) [2];
2. an approach based on the link with an Instruction Set Simulator [3].

The rest of the paper is organized as follows. Section 2 gives an overview of related work. Section 3 describes in detail the proposed methodology and one possible implementation scheme. Section 4 shows some experimental results. Section 5 provides some conclusions and an overview of future extensions and improvements.

2 Related Approaches

The main techniques for software performance estimation fall into four groups:

1. using a cycle-accurate ISS together with a hardware simulator, and filtering the information that is passed between them (e.g., by suppressing instruction and data fetch-related activity in the hardware simulator) [4, 5];

2. compiling the software description and annotating the generated control flow graph (CFG) with information useful for deriving a cycle-accurate performance model (e.g., considering pipeline and cache) [6, 7];
3. trying to guess the compiler optimizations and annotating the original C code with timing estimates [2];
4. using a set of linear equations to implicitly describe the feasible program paths [8].

The first approach is precise, but suffers from a low simulation speed, and requires a detailed model of the hardware and the software. Performance analysis can be done only after completing the design, and hence it is very difficult to modify architectural choices, such as the type of processor, the choice of peripherals, and so on.

The second approach is based on analyzing the code generated for each basic block in the program, trying to incorporate information about the optimization performed by an actual compilation process. This can consider register allocation, instruction selection and scheduling, and so on. In our approach we partially use this scheme, and couple it with a high-level co-simulation framework.

The third approach has the advantage of not requiring a complete design environment for the chosen processor(s), since the performance model is relatively simple (an estimated execution time on the chosen processor for each high-level language statement). However, it cannot consider compiler and complex architectural features (e.g., pipeline stalls due to data dependencies). The method cannot be applied easily to unrestricted C code, but provides good results on code with a very simple structure (e.g., without loops and recursive procedure calls) [2, 6].

The fourth approach has the advantage of not requiring a simulation of the program, and hence can provide conservative worst-case execution time information. However, so far it has been applied only to single programs, and not to multi-tasking environments common in embedded systems.

Sometimes it is also possible to use mixed approaches like in [9], where a software estimation methodology tries to approach each step in the analysis with the best methods currently known is presented.

Our approach fits in the second category, thus this one will be described more in detail below.

In [6] a program timing analyzer for control applications is presented. Given a task written in a subset of C, a compiler constructs the CFG and generates assembly code that is further translated into an executable file. An instruction-level timing analysis is then performed on the CFG and the assembly code for each basic block. The results are labels on the CFG, representing the description of the pipeline state at the entry and exit of each basic block. Data flow analysis is also performed in order to predict data and instruction cache performance.

In [7] a compiled hardware/software co-simulation is presented. This approach is based on generating a C program from the target binary code, and then compiling it on the host enviroment for the co-simulation. This simplifies the translation process with respect to the classical binary-to-binary approaches and improves its portability. It differs from the classical interpreted ISSs, because it translates each target assembly instruction into one or more host instructions, thus eliminating the fetch and decode steps, and resulting in a faster simulation. Moreover, by using the C code as the intermediate format for software simulation, and a behavioral C model of the hardware, it is possible to use a standard source level debugger to debug both hardware and software.

Our approach inherits features of both the approaches described above. We construct a C simulation model by using a modified back-end of the GCC compiler. This allows us to solve some drawbacks of [7], where the CFG of the program is not available and must be re-constructed from the final executable. This is a difficult process, especially in the presence of sophisticated compiler optimizations.

By embedding a C compilation suite in a co-design tool, we also avoid the need to purchase a specific compiler for each target processor. We have identified GCC as an interesting compilation suite for our purposes due to the fact that it has been ported to almost every embedded processor and also due to its good optimitazion capabilities on several existing architectures.

Anyway, it is important to underline that our approach does not depend on the choice of the compiler.

As in [6], we perform a pipeline analysis for each basic block in the program, and we use a caching mechanism to avoid recomputing timing information within each basic block at every execution.[1]

We also provide the user with the possibility to examine the performance of the code with different compiler optimizations, that can be specified on a task-by-task basis by attaching parameters to tasks in our co-simulation framework. To the best of our knowledge, this is the first software estimation methodology where a codesign environment is tightly integrated with a compiler.

3 The integration of the GCC suite in the POLIS framework

A prototype of the proposed software performance methodology has been implemented for the MIPS R3000 architecture, using the GCC compiler and the POLIS co-design environment.

In POLIS [1], the system is described using a formal behavioral model based on a network of communicating entities called Codesign Finite State Machines (CFSMs). The user can map each CFSM to either hardware or software, choose the processor type and cache architecture, estimate the performance and evaluate each mapping with little effort. Finally, a hardware or software implementation, including the real-time operating system (RTOS), is synthesized by POLIS.

The simulation flow of the methodology proposed in this paper is shown in Figure 1.

After the network of CFSMs has been manually partitioned into hardware and software, the very same C model (synthesized by POLIS from the CFSM) is used for both hardware and software simulation. The only difference is the mechanism used to synchronize the various CFSMs. Hardware CFSMs operate concurrently, and require one clock cycle to execute a transition. Software CFSMs require a variable number of clock cycles, as determined by clock cycle counting code inserted by POLIS in the C model. Moreover, their operation is coordinated by a scheduler modeling the RTOS used in the final implementation.

[1] Of course, this involves some accuracy trade-off when instruction timing is data-dependent, as in some multiplication, division and string manipulation instructions.

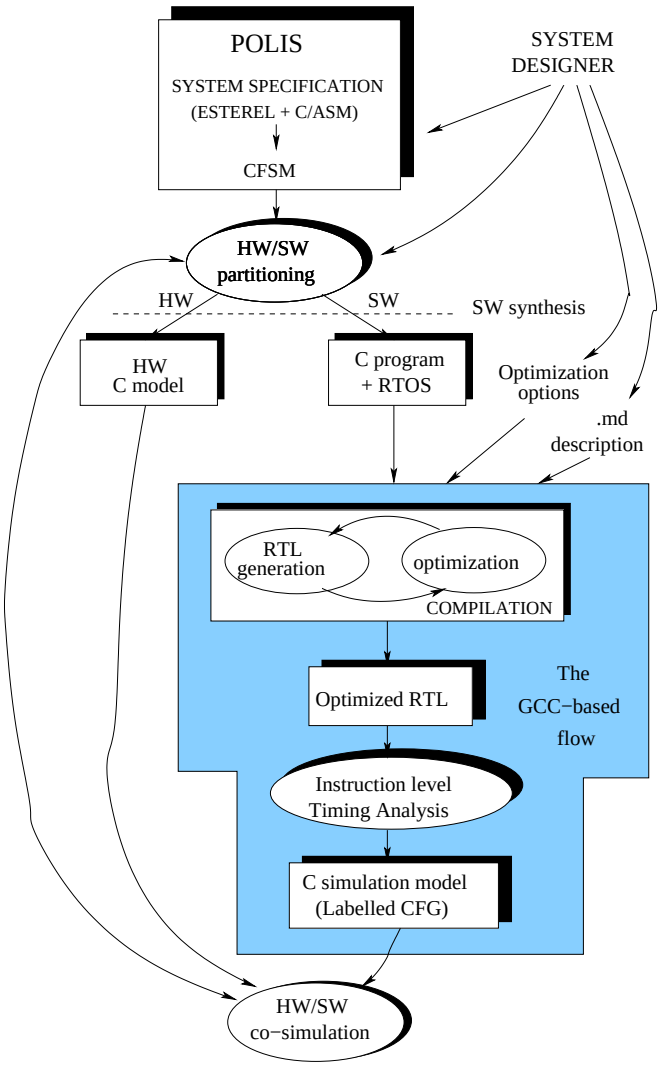

Figure 1: The simulation flow.

In [2], software performance estimates were obtained using a fixed cost for each synthesized C code statement on a given processor (the cost is obtained by executing a set of benchamrks on a cycle-accurate model of the processor). In [3], an ISS-based refinement scheme measures (instead of estimating) the performance of each basic block in the C model, by using a cycle-accurate ISS.

In this paper we added to the original POLIS flow the shaded part of Figure 1. The RTL generation and the optimization is part of the normal GCC compilation process which we do not interfere with. The optimization step can be controlled through a bunch of parameters given by the user and that will be described later.

The optimization phase ends producing an optimized internal RTL structure. We then use this intermediate representation to perform an instruction-level timing analysis and generate a C simulation model that includes both *functional* information (the assembler-like C code) and *timing* information (the added code for clock cycle counting). This model can then be used in the POLIS co-simulation framework without any change.

Note that in POLIS C code synthesized from a CFSM can co-exist with hand-written C code. In this approach (unlike [2]) we can estimate the performance of both kinds of code uniformly.

A significant difference with respect to the performance estimation methodology described in [2] is that now we need a different simulation model for each processor choice and each compiler option combination. This is not a serious drawback, as the model is generated automatically on the fly by the co-simulation environment, but it means that evaluating different processors requires more time in this case.

```
#define  detect_e_RESET_to_z_BELT_0  \
         (*(frozen_inp_events[proc]+0) & 1)
....
v__st_tmp = v__st;
startup(proc);
if (detect_e_RESET_to_z_BELT_0) {
   goto L16;
}
```

```
sb      $2,v__st_tmp.2
jal     startup
lw      $2,proc
#nop
sll     $2,$2,2
lw      $2,frozen_inp_events($2)
#nop
lbu     $4,0($2)
#nop
andi    $2,$4,0x0001
.set    noreorder
.set    nomacro
bne     $2,$0,$L16
andi    $2,$4,0x0004
```

```
DELAY(sb);        v__st_tmp = r2;
DELAY(jal);       // startup(proc); deferred
DELAY(lw+nop);    r2 = proc;
                  startup(proc);
DELAY(sll);       r2 = r2 << 2;
DELAY(lw+nop);    gm_p = &frozen_inp_events+r2;
                  r2 = *gm_p;
DELAY(lbu+nop);   gm_p = 0+r2;
                  r4 = *gm_p;
DELAY(andi);      r2 = r4 & 0x0001;
DELAY(bne);       _jcond = (r2 != r0);
                  // if (_jcond) goto L16; deferred
DELAY(andi);      r2 = r4 & 0x0004;
                  if (_jcond) goto L16;
```

Figure 2: From top to bottom: C code, assembly code and the C simulation model.

Figure 2 shows an example of a little portion of C code generated by POLIS, the resulting assembly code and the corresponding C code used as simulation model. This is mainly composed of DELAY macros and a behavioral part.

The DELAY macros are used to accumulate clock cycles during execution. They receive as argument an arithmetic expression composed of assembly instruction names. A global variable, that represents the time elapsed from the beginning of the simulation, is accordingly updated.

The behavioral part is an assembler-level C code that reconstructs all the functionality of the module. It is possible to see the actual machine registers, references to memory and control instructions. This code is as close as possible to the assembler semantics, but it is not possible to mantain an exact one to one correspondence.

For the simulation, the generated C code both DELAY macros and behavioral gets compiled once again on the host machine (typically different from the target machine for which the assembler was generated) and then executed. In an auxiliary header file we extracted the delays associated in the `.md` file to each assembly instruction. This choice results in a faster simulation in respect to an interpretive simulation where for example the actual instruction fetches are performed.

The drawback of this approach is that it is not possible to mantain the total separation between functional and delay information. For example we have to defer some control instructions in order to maintain the original functionality. This is necessary because we have to undo delay slot fillings that expose some instructions to the pipeline behavior in order to exploit the advantages of a pipelined execution.

3.1 The GCC machine description format

Processor-dependent information for the GCC compiler is mostly stored in a machine description file (`.md`) [10].

This is a textual description composed of instruction definitions and instruction attributes. The instruction definitions define the instruction set of the target machine as RTL (Register Transfer List) expressions. RTL is the intermediate representation on which most of the compilation algorithms operate. RTL uses five kinds of objects: expressions (RTX), integers, wide integers, strings and vectors. Each instruction pattern is defined using an RTL expression called `define_insn`, containing four or five operands:

1. an *optional name*;
2. the *RTL template* that shows the effect of the instruction on the processor state;
3. the *condition* (a C expression) that is used (in addition to the RTL) to decide whether some fragment of compiled code matches this pattern;
4. the *output template* determines how to generate assembler code. When simple general substitution is not general enough, a piece of C code to compute the output can be used. This is the only part of the `.md` file that we had to modify in order to produce C instead of assembler;
5. an optional vector with the values of attributes (e.g., latency, delay slots, ...) for matching instructions.

3.2 User control of optimization parameters

One of the main advantages of our proposed methodology is to give the user a tight control over the compiler optimization capabilities, on a CFSM by CFSM basis. For this reason, the designer can specify the parameters shown in Table 1 (corresponding to GCC compilation options) for each CFSM in the netlist describing the embedded system specification.

For example, the OPTIMIZE parameter selects whether to optimize the code for speed or size. The OPTIMIZATION_LEVEL parameter selects the level of optimization (higher numbers correspond to higher optimizations). The LOOP_UNROLLING parameter is used to enable/disable the loop unrolling optimization (to improve pipeline performance). The user is free to add or change parameters. These are passed as pairs of name and action to the compiler. The POLIS environment, based on UNIX `make`, automatically regenerates each simulation module when either its functional specification (CFSM model) or compilation options are changed.

```
(define_insn "addsi3_internal"
  [(set (match_operand:SI 0
         "register_operand" "=d")
        (plus:SI (match_operand:SI 1
                  "reg_or_0_operand" "dJ")
                 (match_operand:SI 2
                  "arith_operand" "dI")))]
  "! TARGET_MIPS16
   && (TARGET_GAS
       || GET_CODE (operands[2]) != CONST_INT
       || INTVAL (operands[2]) != -32768)"
  "addu\\t%0,%z1,%2"
  [(set_attr "type"     "arith")
   (set_attr "mode"     "SI")
   (set_attr "length"   "1")])
```

Figure 3: A section of a `.md` description.

parameter name	definition
OPTIMIZE	SPEED, SIZE
OPTIMIZATION_LEVEL	0-3
LOOP_UNROLLING	0= not enabled, 1= enabled
INLINE_FUNCTIONS	0= not enabled, 1= enabled
CALLER_SAVES	0= not enabled, 1= enabled
FAST_MATH	0= not enabled, 1= enabled

Table 1: Optimization options.

3.3 Using the C simulation model for implementation

The accuracy of our approach depends on the ability to use the same "structure" for the simulation model and for the final executable that will be loaded on the target processor. Unfortunately, each compiler can perform different kinds of optimizations on the code, e.g., by using different register allocation strategies. This problem can be solved in two different ways, depending on the requirements of the final implementation environment:

1. if GCC is a suitable compiler for the target machine, it is enough to compile the C code with the same options when generating the simulation model and when generating the final executable. This can be easily achieved automatically, by a judicious use of automatically generated `Makefile`s.
2. if GCC cannot be used for the final compilation (e.g., due to company policy, availability of development tools such as debuggers and emulators, limited optimization capabilities of GCC on the selected architecture, ...), the C simulation model generated by our modified GCC can be compiled with another compiler as it preserves the exact functionality of the original C code. If a poorer compiler is used we can hope that it will not *undo* any of the optimizations (it will make good use of them), while a better compiler is free to further improve the quality of the final code. In the latter case we can expect less precise estimations.

task	pixie	macromodeling	GCC-based
ODOMETER	42	72	44
BELT	41	76	40
FUEL	58	93	60

Table 2: Unoptimized code measurements.

task	pixie	macromodeling	GCC-based
ODOMETER	40	72	39
BELT	38	76	36
FUEL	53	93	51

Table 3: Optimized code measurements.

4 Performance Simulation and Results

We performed several experiments to evaluate both the accuracy of our approach and the simulation speed. We used a reasonably complex embedded system distributed with POLIS, a dashboard controller.

Table 2 and 3 report the results obtained for the average number of cycles/task obtained for three modules of the dashboard. The first one is for the case of non optimized code, while the second one is for the case of optimized code. In both tables are reported respectively the results obtained with the `pixie` tool profiler for the MIPS architecture (considered a golden reference for this experiment), the macromodeling approach of POLIS [2] and the GCC-based approach.

The following conclusions can be drawn from these tables:

- as expected the impact of compiler optimizations is significant and the macromodeling approach, that cannot take them into account, results in a big discrepancy (an average error of 85%);
- the GCC-based estimation, features a good accuracy in both cases (an average error of less than 4% for both optimized code and unoptimized code) because it reflects the compilation performance.

Regarding the simulation speed we can say that the GCC-based estimation features an average of only 8% overhead with respect to POLIS macromodeling. These results were obtained by using the `prof` utility on a SUN SPARC-20 workstation.

5 Conclusions and Future Work

A new compilation-based software performance analysis method has been presented. We believe that this is the first approach that combines a state-of-the-art optimizing compiler with a high-level co-simulation and co-design methodology.

In the future, we are planning to integrate our work with a front-end that is currently under development in the GCC group. This will eventually allow a processor model developer to generate a machine description from a template file that is less dependent on the internal compiler structure than the `.md` files.

Acknowledgements: The authors would like to thank Prof. Luciano Lavagno for his precious comments and suggestions. We also thank all the members of the Software Estimation Group in the Felix Initiative of Cadence Design Systems: Ed Harcourt, Camille Batarekh, Marek Ryniejski, Doug Dunlop, Neeti Bhatnagar and Soumya Desai, for their help with defining the problem and outlining solutions. Finally we would like to acknowledge Jose Manuel and Francisco Moya from Universidad Politecnica de Madrid for their useful hints in using the GCC suite.

References

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jureska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA., 1997.

[2] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient software performance estimation methods for hardware/software codesign," in *Proc. Design Automation Conf.*, pp. 605–610, Jun. 1996.

[3] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, "Software timing analysis using hw/sw cosimulation and instruction set simulator," in *Proc. Int. Workshop on Hardware/Software Codesign*, pp. 65–69, Mar. 1998.

[4] *Mentor Graphics Seamless CVE Home Page*. http://www.mentorg.com/seamless/.

[5] *Synopsys' Eagle Home Page*. http://www.synopsys.com.tw/products/hwsw/eagle_ds.html.

[6] F. Stappert, "Predicting pipelining and caching behaviour of hard real-time programs," 1998. C-LAB internal document, Furstenalle 11, D-333102 Paderborn, Germany.

[7] V. Zivojnovic and H. Meyr, "Compiled hw/sw co-simulation," in *Proc. Design Automation Conf.*, 1996.

[8] S. Malik, M. Martonosi, and Y. Li, "Static timing analysis of embedded software," in *Proc. Design Automation Conf.*, pp. 147–152, Jun. 1997.

[9] R. Ernst and W. Ye, "Embedded program timing analysis based on path clustering and architecture classification," in *Proc. Int. Conf. Computer-Aided Design*, pp. 598–604, Nov. 1997.

[10] R. M. Stallman, *Using and Porting GNU CC*. http://www.delorie.com/gnu/docs/gcc/gcc_toc.html.