Compilation-based software performance estimation for system level design

Compilation-based Software Performance Estimation for System Level Design

Mihai T. Lăzărescu
Cadence Design Systems, Italy
mihail@cadence.com

Jwahar R. Bammi
Cadence Design Systems, USA
bammi@cadence.com

Edwin Harcourt
Cadence Design Systems, USA
harcourt@cadence.com

Luciano Lavagno
Università di Udine, Italy
lavagno@uniud.it

Marcello Lajolo
NEC USA C&C Research Labs
lajolo@ccrl.nj.nec.com

Abstract

The paper addresses embedded software performance estimation. Known approaches use either behavioral simulation with timing annotations, or a clock cycle-accurate model of instruction execution (e.g., an instruction set simulator.)

We propose a hybrid approach, that features both the high simulation speed and flexibility from the former approach and the awarness of compilation optimizations and processor features of the later. The key idea is to translate the assembler generated by a target compiler to an "assembler-level," functionally equivalent, C code. This code, annotated with timing and other execution related informations, is used as a very precise, yet fast, software simulation model. The approach is used in Cadence VCC, a system-level design environment.

We report a comparison of several known approaches, the description of the new methodology, and experimental results, that show the effectiveness of the proposed method. We also propose several improvements.

Keywords: *real-time systems, timing analysis, compilation, architecture modeling.*

1. Introduction

Today, high performance IC technologies combine ever increasing computing power with complex integrated peripherals and large amounts of memory at decreasing costs. The software content of the embedded systems grows exponentially, mostly from the migration of application-specific logic to application-specific code, to cut down products costs and time to market.

Short product life cycles and customization to niche markets force designers to reuse not only building blocks, but entire architectures as well. A new architecture is analyzed for appropriateness and efficiency to different applications or behaviors to determine its market size. Every time new features are added, the architecture should be re-analyzed to ensure it provides the right timing and support. Using efficient system development tools can significantly alleviate this problem.

Once mapped onto an architecture, the behavior can be annotated with estimated execution delays. The delays depend on the implementation type (hardware or software) and on the performance and interaction of the architectural elements (e.g., IC technology, access to shared resources, etc. for hardware, clock rate, bus width, Real-Time scheduling and CPU sharing, etc. for software.) These estimates should be accurate enough to allow making high level decisions such as what behavior needs to be implemented in hardware, how to architect the software in terms of threads, and what RTOS to use. For real-time embedded systems, that have strict timing constraints, the design tools are expected to assist the designer with *accurate* timing simulations at early stages of system definition. High level performance estimation coupled with a fast co-simulation framework seems a suitable solution to forestall performance problems in embedded system design.

Providing early good timing information for the hardware/software co-simulator before designing detailed hardware and software is a very difficult problem, especially for the software side. Architectural enhancements can rapidly obsolete established good software estimation techniques. This goal was pursued through various methods (e.g., [10, 8, 11]), but none of these is suitable for the function/architecture co-design methodology. They generally target worst case execution time analysis for a single program. These approaches are not suitable for embedded systems, composed of multiple tasks accessing common resources, whose dynamic activation can significantly modify

each other's execution path.

Our research effort aims at developing techniques and tools to accurately evaluate the performance of a system at different levels of abstraction. The evaluation must be done dynamically, in a simulation environment, to capture run-time task interactions. It also should be fast to allow the exploration of several architectural mappings in search for the best implementation. Tunable models, where the designer can trade accuracy for speed, would do the best.

In this paper, we target a system-level design approach pioneered in the Felix initiative of Cadence Design Systems, Inc. [5]. Felix enforces a separation between system behavior and architecture. In this way, the system designer focuses first on system behavior, then looks for a suitable architecture to implement it.

The rest of the paper is organized as follows. Section 2 introduces the performance estimation problem and gives an overview of the related work. Section 3 describes our compilation-based approach and discusses solutions to some related problems. Section 4 presents the results obtained for various examples. Section 5 concludes the paper.

2. Motivation and background

2.1. Overview

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

1. filtering the information that is passed between a cycle-accurate ISS and a hardware simulator (e.g., by suppressing instruction and data fetch-related activity in the hardware simulator) [1, 2];

2. annotating the control flow graph (CFG) of the compiled software description with information needed to derive a cycle-accurate performance model (e.g., considering pipeline and cache) [12, 14];

3. annotating the original C code with timing estimates trying to guess compiler optimizations [13];

4. using a set of linear equations to implicitly describe the feasible program paths [10].

The first approach is precise but slow and requires a detailed model of the hardware and software. Performance analysis can be done only after completing the design, when architectural choices are difficult to change.

The second approach analyzes the code generated for each basic block and tries to incorporate information about the optimization performed by an actual compilation process. It considers register allocation, instruction selection and scheduling, etc.

The third approach uses an estimated execution time on the chosen processor for each high-level language statement, thus it does not require a complete design environment for the chosen processor(s). However, it cannot consider compiler and complex architectural features (e.g., pipeline stalls due to data dependencies.) The method cannot be applied easily to arbitrary C, either [13, 12].

The fourth approach provides conservative worst-case execution time information without requiring a simulation of the program. However, it has been applied only to single programs so far, and not in multi-tasking environments, so common in embedded systems.

Mixed approaches can be used under some circumstances. For example, [8] tries to approach each step in the analysis with the best currently known methods.

In [12], a compiler constructs the CFG, generates assembly code and an executable for a task written in a subset of C. An instruction-level timing analysis is performed on the CFG and the assembly code. The CFG is annotated with the pipeline state, and data and instruction cache performance is predicted using data flow analysis.

In [14], a compiled hardware/software co-simulation is presented. A C program is generated from the target binary and compiled on the host for co-simulation. The translation is simplified with respect to binary-to-binary approaches and improves the portability. It is also possible to use a standard source level debugger to debug both hardware and software.

2.2. Proposed solution

Our previous work in this area [13] was based on source code analysis with timing annotations. That approach had several limitations, as discussed below. In this paper we discuss an assembler-level timing analysis methodology that solves, as shown by our experimental results, most of those limits, while keeping an extremely efficient execution time.

Estimation at the source code level has serious problems taking into account compiler optimizations. Typical optimizations performed by a compiler are constant propagation, dead-code elimination, common sub-expression factoring, loop optimizations (e.g., global variables copied in machine registers, extraction of loop independent expressions), strength reduction (e.g., a multiplication by a constant is replaced by shifts and additions.) A performance estimation that is not aware of these may be very loose in several cases. For example, in figure 1, all but the last two lines inside the loop depend on variables that do not change inside the loop, so that they can be computed outside the loop. This reduces the number of multiplications from 1000 to only 303. These cases are fairly common when the code is generated through automatic synthesis.

A machine-independent C-to-C optimization to predict

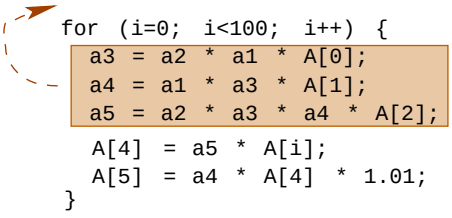

Figure 1. An example of loop optimization.

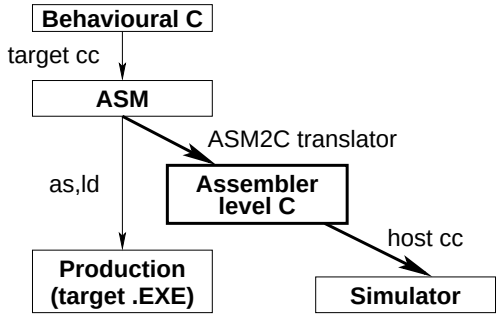

Figure 2. Simulation preparation flow.

at least the most common optimization compiler optimizations is a trade-off between source level and assembler-level estimation. This is not straightforward though, since some optimizations are effective for specific architectures only.

Thus we decided to perform the estimation after compilation. Our approach, that is closely related to compiled-code co-verification techniques, inherits features of both approaches described in [12] and [14]. An annotated C simulation model is generated from the assembler output of the compiler and coupled with a high-level co-verification framework. Although the performance annotations reflect the compiler optimizations, the approach itself does not depend on the choice of the compiler. As in [12], a pipeline analysis is performed at annotation time for each basic block to speed-up simulation execution.

3. Compilation-based software estimation

The flow proposed in this paper is shown in figure 2. The high-level C code of a given software task of the system is compiled with the target C compiler. The output assembler is translated to an assembler-level C co-simulation model, annotated with timing information. The accuracy is high since all the architectural effects (instruction scheduling, register allocation, addressing modes, memory accesses...) are visible at this level. On the other path, the same assembler is used to generate the executable that will run in the target environment.

Our software estimation technique supports also co-simulation. Assembler-level *translated* C models can be mixed with functional *non-translated* models, possibly annotated using less precise techniques or by hand. This is a flexible mechanism to incorporate refined delay estimates into the performance model as they become available. Library functions can be pre-characterized in terms of delay model, making our approach more efficient with respect to an ISS-based solution which, for example, has to interpret any function in the C library every time it is called.

The basic assumptions needed to generate an accurate simulation model using this method are:

- the input program has been optimized by the target compiler. Except for hardware optimizations, made by the target architecture at run time, no other optimization will be made (e.g., by the assembler);
- the optimizations made at run time by the target architecture (e.g., register renaming) are known. Model efficiency is best if they are data-independent;
- the input for the translator is generated by the same compiler that will be used for the target executable.

Figure 3 shows a fragment of behavioral C code, the resulting assembler, and the C simulation model. The model is composed of timing annotations (DELAY macros in figure 3) and behavioral part. DELAY accepts a sequence of assembly mnemonics as argument, and accumulates timing information during execution. The behavior is an assembler-level C, that references directly the host memory but uses emulated target registers and stack. These, as well as arithmetic operations, etc., are mapped on the hardware host resources by the compiler of an ISA.

Function and timing cannot always be split. For example, in the model control instructions may be deferred. In the sequel we will discuss the modeling of delay slots, indirect jumps, calling conventions, condition codes, and memory access. The target processor is a RISC MIPS R3000, but the same considerations apply to most modern processors (we will comment on differences due, e.g., to a CISC architecture wherever appropriate.)

3.1. Delay slots

The delay slots allow pipeline performance exploitation exposing the pipeline behavior. MIPS R3000 has one delay slot after all jump and load instructions [9]. Since it's not fully interlocked,[1] the target of the load can not be used in the delay slot. Thus, the loads are modeled with a `NOP` inserted in the delay slot whenever necessary.

[1] MIPS is an acronym for **M**icroprocessor without **I**nterlocking **P**ipe **S**tages. This absence of interlocks also characterizes modern VLIW architectures.

```
v__st_tmp = v__st;
startup(proc);
if (frozen_inp_events[proc][0] & 1)
  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
.set    reorder
.set    macro
```

```
DELAY(sb);    v__st_tmp = R2;
DELAY(jal);   // startup(proc); deferred
DELAY(lw);    R2 = proc;
              startup(proc);
DELAY(nop);
DELAY(sll);   R2 = R2 << 2;
DELAY(lw);    R2 = *(&frozen_inp_events+R2);
DELAY(nop);
DELAY(lbu);   R4 = *(0+R2);
DELAY(nop);
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 3. From top to bottom: C code, assembly code and the C simulation model.

Since the jump instructions may alter the program flow, the C model executes the instruction in the delay slot before the branch. For example, a branch instruction like:

```
    sub     $2,$1,$3
    j       $31             ; return on MIPS
    add     $2,$2,3
```

preserves the behavior if modeled in C as follows:

```
        R2 = R1 - R3;
        // return; deferred
        R2 = R2 + 3;
        return;
```

3.2. Indirect jumps

Indirect jumps are the most difficult to model. A runtime C construct (a `switch` statement) is necessary to translate physical to symbolic addresses.

All possible destinations of the indirect jumps are associated a unique symbol (for example, by annotating every instruction of the assembler source with a unique label and assemble it to an object file.) The physical address – symbolic label pairs are used to build a table for the C preprocessor as follows:

```
    #define LSW1    0x0001
    #define LSW2    0x0002
    #define LSW3    0x0003
    ...
```

The same set of labels is used in the selector of the `switch` statement in the C model, each translated construct being preceded by the corresponding label:

```
                        PC = LSW1;
                        for (;;)
                          switch (PC) {
  . . .                   . . .
LSW1:                     case LSW1:
  lw    $4,proc             R4 = *(int *)(&proc);
LSW2:                     case LSW2:
  j     $4                  // deferred jump
                            VCC_lbv = R4;
LSW3:                     case LSW3:
  subu  $sp,$sp,24          SP = SP - 24;
                            PC = VCC_lbv; break;
  . . .                   . . .
                          }
```

For all instructions except jumps the program flow falls smoothly through the `switch` cases. The indirect jump behavior saves the destination address, executes the delay slot instruction, then sets the switch selector (`PC`) to the destination address and breaks out of the `switch`. The endless loop re-enters the `switch` at the jump target. The only overhead is the update of `PC` and the execution of the `switch` at each indirect jump.

This is a general solution, with applicability restricted to small pieces of code. The size of the `switch` in the simulation model may easily overflow the host compiler capacity or force it to turn off optimizations. When the target compiler code generation strategy is known, more efficient indirect jump handling can be used. One such case is the jump table produced for *switch* statements [4], which may be reconstructed to C `switch` statements. Another case is for procedure return in many RISC processors, such as the MIPS and ALPHA [7], which can be translated to C `returns`.

3.3. Calling conventions

Different architectures use different calling techniques: using the stack, registers, register windows, etc. For example, the SPARC uses overlapping windows, whereas some MIPS compilers perform an inter-procedural register allocation. Interfacing translated and non-translated functions is challenging, since the host and target machines typically use different calling conventions.

There are three cases, depending upon whether caller and callee are translated or not:

1. Both caller and callee are translated or not. No special actions performed;

2. Only the caller is translated. The actual arguments may be in emulated registers, specific stack locations, etc. They are used into the modeled C function call.

   For example, MIPS uses registers $4-$8 for the first 4 integer-size arguments and $2 for an integer return. If the callee arguments, for example, are two integers:

   ```
   int callee_function(int a, int b);
   ```

   the call statement in the generated C code must be:

   ```
   R2 = callee_function(R4, R5);
   ```

   where R2, R4, and R5 are emulated registers.

3. Only the callee is translated. A callee model prologue converts from the host to the target calling convention:

   ```
   void f(int a, int b) {
     R4 = a;  /* Conversion from host to */
     R5 = b;  /* target calling conv.    */
     /* Body of the function */
   }
   ```

   If it is also called by translated functions, its code is duplicated with no prolog, and its calls are as in case 1.

   ```
   void my_f(void) {
     /* Body of the function */
   }
   ```

3.4. Condition codes

The condition codes are used in branch decisions much less than they are set. There is a known problem in modeling this efficiently [6]. A data flow analysis flags only the condition code changes that may be used by a conditional branch instruction and only the flagged condition codes updates are output.

3.5. Memory access

The simulation model accesses directly the host memory. Our method supports a mix of translated and non-translated functions who share variables.

Uninitialized static, local, and external symbols are converted to arrays of `char`s in the C model, and extended to the next host word boundary. Appropriate casts are provided in all translated statements for correct memory access. The assembler code directly provides the instructions for accessing `struct` fields, and vector elements, dereferencing pointers, etc., starting from the base symbol address.

We use arrays of `char`s to have a base type size of one byte and facilitate the translation of indirect addressing. The offset in assembler is measured in bytes, thus we need a base type size of one byte in the C model also.

If a program includes both translated and non-translated functions, then the target and host machine need to have the same representation for the basic storage types: `int`s, `short`s, the same representation for the floating point types, and the same 'endianness'.

4. Experimental results

We report experimental results on two examples taken from very different domains – a car dashboard controller and a classical bubble sort algorithm. The dashboard controller is a control-dominated reactive multi-tasking application with both hard and soft deadlines.[2] The bubble sort algorithm is a data dominated application.

Figure 4 reports the relative error of the estimated execution time on a MIPS R3000 architecture, with respect to a cycle accurate profiler. Four modules of the *dashboard* application (`belt`, `odometer`, `fuel`, and `speedometer`) and the bubble sort algorithm were measured. The new method is compared against the source-level estimation distributed with the POLIS tool [13].

The source annotation ignores the level of optimization of the compiler, hence the errors are high. Since the compilation-based approach reflects all compiler optimizations as well as many local architectural effects, like pipeline stalls, its error is very low.

The simulation using the proposed approach runs in the worst case three times slower than the model generated using source code annotation, thus achieving a performance close to 13 million simulated clock cycles per CPU second (without cache simulation) on a 500MHz Pentium II workstation.[3] This performance was achieved within an embedded system design environment [5].

5. Conclusions and Acknowledgments

System functionality is often implemented in software, yet estimating software performance of embedded multi-

[2] The interested reader can find additional information on this system and on its specification within the POLIS environment in [3].

[3] The $40\times$ performance loss with respect to real time is due to the simulation overhead, for task synchronization, etc.

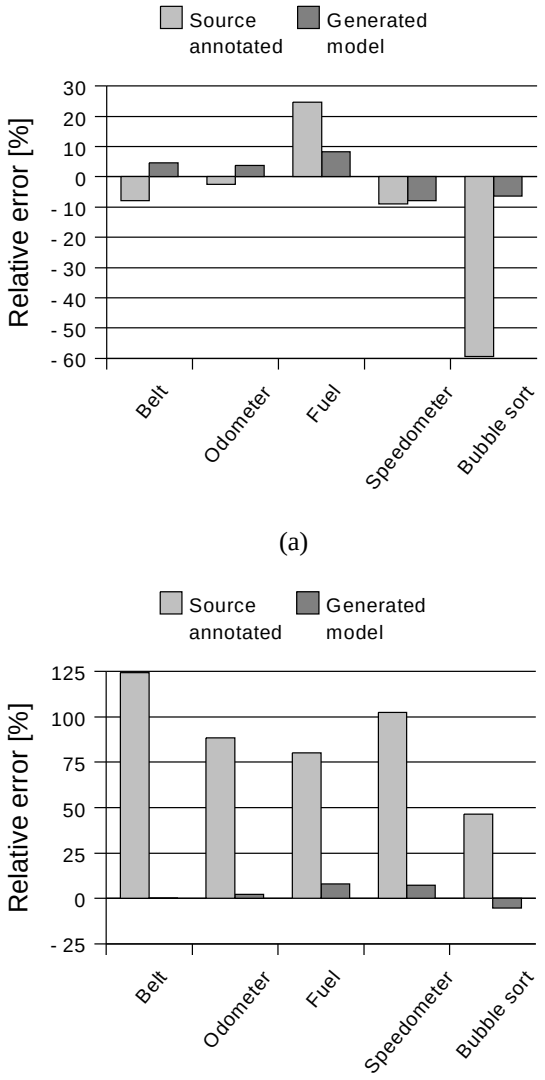

Figure 4. Relative estimation error of the execution time for unoptimized code (a) and optimized code (b), for two estimation techniques.

tasking reactive systems is still a difficult problem. Reasonably accurate performance estimation is needed for estimating the effect of the pipelines, multiple instructions issue, code and data caches, and memory hierarchies. The work presented in this paper attempts to analyze the time performance of software as accurately as possible, while still achieving a high simulation speed (at least one order of magnitude faster than a cycle-accurate ISS [14].)

We are planning to apply this technique also to VLIW (Very Long Instruction Word) processors. This type of processors presents a synergy between the compiler design and the hardware design. A VLIW compiler performs complete static code scheduling avoiding runtime pipeline interlocks, so that the code can be executed without having to take any control decisions at runtime. This makes our method very well suited for these architectures.

References

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

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

[3] 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.

[4] C. Cifuentes and M. V. Emmerik. Recovery of jump table case statements from binary code. *Proceedings of the International Workshop on Program Comprehension*, pages 192–199, May 1999.

[5] P. Clarke. Felix tools pushed in research project. *Electronic Engineering Times*, Oct. 1998. See http://www.eetimes.com/news/98/1029news/felix.html.

[6] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profi ling. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 22(1):128–137, May 1994.

[7] T. Conte and C. Gimarc. *Fast Simulation of Computer Architectures*. Kluwer Academic Publishers, 1995.

[8] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classifi cation. In *Proc. Int. Conf. Computer-Aided Design*, pages 598–604, Nov. 1997.

[9] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, NJ, 1992.

[10] S. Malik, M. Martonosi, and Y. Li. Static timing analysis of embedded software. In *Proc. Design Automation Conf.*, pages 147–152, Jun. 1997.

[11] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991.

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

[13] K. Suzuki and A. Sangiovanni-Vincentelli. Effi cient software performance estimation methods for hardware/software codesign. In *Proc. Design Automation Conf.*, pages 605–610, Jun. 1996.

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