POLITECNICO DI TORINO
Repository ISTITUZIONALE

Software performance estimation strategies in a system-level design tool

Original
Software performance estimation strategies in a system-level design tool / J. R., B., E., H., W., K., Lavagno, L., Lazarescu, M.T.. - ELETTRONICO. - (2000), pp. 82-86. (CODES '00 San Diego, California, USA) [10.1145/334012.334028].

Availability:
This version is available at: 11583/1667466 since: 2018-10-30T15:14:31Z

Publisher:
ACM

Published
DOI:10.1145/334012.334028

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript

(Article begins on next page)

12 October 2026

Software Performance Estimation Strategies in a System-Level Design Tool

Jwahar R. Bammi, Edwin Harcourt
Cadence Design Systems
bammi@cadence.com
harcourt@cadence.com

Wido Kruijtzer
Philips Research Laboratories
wido.kruijtzer@philips.com

Luciano Lavagno, Mihai T. Lazarescu
Politecnico di Torino
lavagno@polito.it
lazarescu@polito.it

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 difficult.

In this paper we describe two approaches to solve software cost and performance estimation problem, and how they are used in an embedded system design environment. A source-based approach uses compilation onto a *virtual instruction set*, and allows one to quickly obtain estimates without the need for a compiler for the target processor. An object-based approach translates the assembler generated by the target compiler to "assembler-level," functionally equivalent C. In both cases the code is annotated with timing and other execution related information (e.g., estimated memory accesses) and is used as a precise, yet fast, software simulation model. We contrast the precision and speed of these two techniques comparing them with those obtainable by a state-of-the-art cycle-based processor model.

1. INTRODUCTION

With the ability to mix processors, complex peripherals, and custom hardware and software on a single chip, full-system design and analysis demand a new methodology and set of tools.

Nowadays, high performance IC technologies combine ever increasing computing power with complex integrated peripherals and large amounts of memory at decreasing costs. It comes as no surprise that the software content of embedded systems grows exponentially. While the system development tool industry has overlooked this trend for years, most estimates place the software development cost at well over half the total development budget for a typical system. The bias towards software development in system-level design arises mostly from the migration of application-specific logic to application-specific code, driven mainly by the need to mitigate product costs and time to market pressures.

Short product life cycles and customization to niche markets force designers to reuse not only building blocks, but entire architectures. The final production cost is often paramount, thus the prime directive is to find the right combination of processor, memory, and glue logic for efficient manufacturing. This means that several candidate architectures must be analyzed for appropriateness and efficiency with respect to different applications or behaviors. The fitness of a new architecture

An important part of the design consists in mapping the behavior (from the specifications) to the architectural blocks (from IP suppliers) in such way that the cost, power consumption, and timing of the system can be analyzed. For the hardware, ASIC companies provide gate-level models and timing shells. For the software, a similar characterization method is expected from system development tools.

When properly separated behavior and architecture may coevolve. As new requirements in the behavior call for changes in the architecture, architecture considerations (e.g., production cost) may lead to behavior modifications. Good system design practice maintains an abstract specification while allowing independent mapping of behavior onto architecture. This is the essence of what has been termed function/architecture co-design [6, 7].

Once mapped, 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, and clock rate, bus width, Real-Time scheduling and CPU sharing, etc. for software.) These estimates should be accurate enough to help make high level choices such as which behaviors should be implemented in hardware or which in software. Once a behavior is mapped to software there may be decisions about how to architect the software in terms of tasks, RTOS scheduling policy, and task priorities.

In this paper we present and contrast two techniques and tools to accurately evaluate the performance of a system, working at different levels of abstraction in order to trade off precision and speed. To capture run-time task interaction the evaluation must be done dynamically – in a simulation environment. Moreover, it should be fast enough to enable the exploration of several architectural mappings in search of the best implementation. We focus mainly on software written in C, because this is the dominant high-level language in embedded system programming. However, the approach can be applied equally well

to other languages, such as C++ or Java. Moreover, the object code-based approach can also be used (with some limitations discussed in [9]) to estimate pre-coded software blocks written in assembler. This is ideal for DSP software blocks whose implementation is commonly assembler.

The rest of the paper is organized as follows. Section 2 introduces the performance estimation problem and overviews related work. Section 3 describes our source-based approach and discusses some of its drawbacks. Section 4 describes a more precise object code-based approach and discusses some of the trade-offs between the two. Section 5 presents the results obtained for various benchmarks and examples. Section 6 concludes the paper.

2. MOTIVATION AND BACKGROUND

The main software performance estimation techniques fall into four groups:

1. filtering 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) [2, 3];

2. annotating the control flow graph (CFG) of the compiled software description with information useful to derive a cycle-accurate performance model (e.g., considering pipeline and cache) [11, 13];

3. annotating the original C code with timing estimates trying to guess compiler optimizations [12];

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 analyses 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. In our object code-based approach we partially use this scheme.

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.) In our source-based approach we extend this method to handle arbitrary hand-written C code, rather than just synthesized code.

The fourth approach has the advantage of not requiring a simulation of the program, hence it can provide conservative worst-case execution time information. However, so far it has been targeted at worst case execution time analysis for a *single program*. Embedded systems, on the other hand, are composed of multiple tasks, accessing common resources, whose dynamic activation can significantly modify each other's execution path or timing behavior (e.g., by changing the state of the cache.)

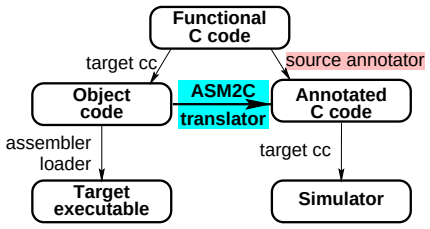

Figure 1: Simulation preparation flow.

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.

3. SOURCE-BASED ESTIMATION

The estimation flows discussed in this paper are shown in figure 1. In this section we focus on the flow depicted on the right, based on direct annotation of the C source. The key idea is that of performing a *partial compilation* of the source code using classical techniques described, for instance, in [4]. A minimum amount of information is retained to estimate the size and execution time of the object code that the target compiler would emit. In this way, one does not need access to the complete development environment (compiler, assembler, debugger, loader, Instruction Set Simulator) for the target processor, since a single "Virtual Compiler" approximates the result of the target compiler.

The *Virtual Instruction* (VI) set that we use as the target of the virtual compilation process includes all the main classes of instructions offered by existing processors. It is based on a *RISC philosophy*, in that it provides a small set of instructions out of which others can be synthesized. The VI set that is currently used by our tool is as follows. We show also, as an example, estimated cycle counts for each VI for the Motorola MCORE processor [1] (the methods for deriving them are discussed below):

LD, ST: load/store a register from/to memory (2 cycles.)

OP.c, OP.s, OP.i, OP.l: simple arithmetic operations on a byte, short word, word, and long word, respectively (1 cycle.)

OP.f, OP.d: single- and double-precision floating point operation (160 and 360 cycles respectively.) Floating-point operations are emulated in software and depending on the algorithm can take various numbers of cycles. Here we used an average number estimated with a cycle accurate ISS (hence the VM method will not be accurate for a program using many such operations.)

MUL.*, DIV.*: multiply and divide operations. See above.

SUB, RET: subroutine call and return (8 cycles.) The total cycle count includes overhead for storing the five non-volatile registers (using LDM for the five register takes 6 cycles) plus the time for a BSR (2 cycles), and vice-versa for return.

IF, GOTO: conditional and unconditional branch (2 cycles.) In general this is accurate for all of the branch instructions except for the "predictive" branch instructions, which take only 1 cycle if the branch not taken. Hence 2 cycles is a worst case estimate.

```
v__st_tmp = v__st;
startup(proc);
if (frozen_inp_events[proc][0] & 1) {
    goto L16;
}
```

```
. . .
v__st_tmp = v__st;
__DELAY(LI+LI+LI+LI+LI+LI+OPc);
startup(proc);
if (frozen_inp_events[proc][0] & 1) {
    __DELAY(OPi+LD+LI+OPc+LD+OPi+OPi+IF);
    goto L16;
}
__DELAY(OPi+LD+LI+OPc+LD+OPi+OPi+IF);
```

```
sb      $2,v__st_tmp.2
jal     startup
lw      $2,proc
sll     $2,$2,2
lw      $2,frozen_inp_events($2)
lbu     $4,0($2)
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);      _R2 = *(&frozen_inp_events+_R2);
DELAY(lbu+nop);     _R4 = *(0+_R2);
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, source-based simulation model, assembly code, and object code-based simulation model.

Each virtual instruction represents a class of instructions on the target architecture. For example, the OP virtual instruction class represents over sixty different MCORE integer-ALU instructions.

The first part of figure 2 presents a C fragment followed by an automatically annotated version of the same file. This annotated version is used for performance simulation and is annotated using the previously described Virtual Machine instructions. The simulation model is mainly composed of timing instructions and the remaining behavioral part. Timing information accumulates during function execution. They receive as argument an arithmetic expression, composed of encoded Virtual Instruction names. Each Virtual Instruction has associated a delay value, taken from the "processor basis file." This value is added to a global variable, that represents the accumulated clock cycles from the beginning of the simulation.

Similar annotations (not shown in the figure) also model memory accesses for instructions fetches and data loads and stores, that are filtered and estimated by a cache model, a bus model and memory model not discussed in this paper.

One key aspect of our approach is that we generate virtual instructions *independent of the target processor*. The only aspect where the processor is taken into account is when evaluating the delay of each virtual instruction. In this way, changing the processor choice for a given piece of C code is simply a matter of changing the basis file used by the simulator. On the other hand, there are disadvantages discussed further below.

3.1 Generation of the Processor Basis file

The processor basis file for each supported CPU contains an estimated cycle count for each Virtual Instruction. This cycle count must take into account an approximate view of the *processor pipeline* (for example, the cycle counts for the MCORE shown above take an optimistic view of it assuming that there are never stalls.) At the same time, it ignores the behavior of the memory hierarchy and system busses. These are taken into account by other components of our simulation framework, and the Virtual Compiler (as well as the object-based estimator described below) only generates *estimated addresses* for instruction and data accesses (not shown in the figures for the sake of simplicity.) Treatment of these additional architecture-dependent delays is however outside the scope of this paper.

Currently we use two techniques in order to derive cycle counts for each VI. The first technique is fairly straightforward, and is generally used to bootstrap and verify the results of the second one. It amounts to reading the processor manual and/or using a cycle-accurate ISS (e.g., for VIs implemented in SW) and filling in a table. The second technique is based on a statistical parameter identification procedure. We compile a set of benchmarks (providing a mix of all the VIs) both using the virtual compiler and using the real compiler. We then simulate the annotated C code and obtain a number of executions for each VI in each benchmark. We also run each benchmark on a cycle-accurate ISS or emulator of the target processor. Now we have a system of linear equations of the form:

$$\begin{aligned} n_{1,1} * c_1 + n_{1,2} * c_2 + \ldots + n_{1,m} * c_m &= N_1 \\ n_{2,1} * c_1 + n_{2,2} * c_2 + \ldots + n_{2,m} * c_m &= N_2 \\ &\ldots \\ n_{n,1} * c_1 + n_{n,2} * c_2 + \ldots + n_{n,m} * c_m &= N_n \end{aligned}$$

where each $n_{i,j}$ is the number of VI's of type j that our simulator dynamically estimates in the execution of benchmark i, each c_j is the (unknown) cost of VI j, and each N_i is the actual cycle count for benchmark i. Note that the system has more equations than unknowns, and hence we actually to solve it by minimizing the error between the predicted and the actual counts.

3.2 Drawbacks of the Virtual Compilation approach

The most important limitations of an estimation made at the source code level are that it is quite difficult to account for potential compiler optimizations and for (CISC-style) instructions that do not fall into any of the VI categories. For example, our source-level estimator does not know how many registers the target processor has. Hence it must estimate which variables are likely to be stored in registers versus in memory. In the current implementation we *optimistically* assume that scalar function arguments and local scalar variables always end up in registers.

Even if the source code analyzer could perform compiler-like optimizations, it would be impossible to guarantee that they

would match those done by the actual target compiler. For these reasons we discuss next a more precise approach, based on performing estimation after compilation and annotating a regenerated C simulation model from the assembler output by the compiler.

4. OBJECT CODE-BASED ESTIMATION

Consider now the flow shown on the left hand side of figure 1. 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 model, that maintains the original behavior, and is annotated with timing information. This timing information is very accurate, since all the architectural effects (instruction scheduling, register allocation, addressing modes, memory accesses...) are visible at this level. The assembler-level C is used as a very precise, yet fast, co-simulation model. On the other path, the very same assembler is used to generate the executable that will run in the target environment.

Another important aspect of our software estimation technique is that it supports a co-simulation where assembler-level *translated* C models can be mixed with functional *non-translated* models (possibly annotated using the source-level technique discussed in Section 3 or by hand.) This feature may become useful when it is not possible to compile the entire application (e.g., some blocks can be linked only as objects and are not available in the source form.)

The possibility to mix pre-compiled and pre-characterized simulation models for library functions can also make our approach more efficient with respect to an ISS-based solution. In that case, for example, any function in the C or mathematical function library has to be interpreted every time it is called, while in our case it could have a faster, hand-written timing model.

The accuracy of the method relies on the fact that the simulation model has the same behavior as the original program. Hence, 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, speculative execution) are known. Model accuracy 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.

However, this may not be possible. The target compiler may not be available at the moment of system architecture definition, or it may not output the assembler file, or a disassembler for the target executable may not be available. In this case we can use another supported target compiler to generate the assembler and the simulation model. Once generated, the model is a portable source by itself, and it can be recompiled with the 'official' target compiler in production stage.

Figure 2 shows a fragment of C code, the associated Virtual Machine simulation model, the assembly code, and the corresponding C code of compilation-based simulation model.

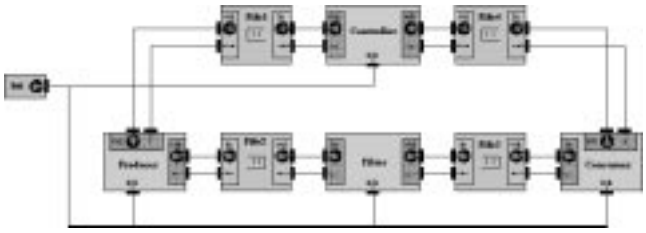

Figure 3: PFC block diagram.

The latter is made of two main parts: the behavioral part and the timing part. The behavioral part is an assembler-level C code that reconstructs the behavior of the function. It makes use of emulated registers, condition codes, and stack, references to host memory and flow control instructions. When the simulation model is compiled by the host compiler, the references to the emulated registers would be ideally reallocated to host registers by the host compiler, so that the simulation model runs fast.

For the simulation, the generated C model is compiled once again on the host machine and executed. This results in a faster simulation with respect to an interpretive (ISS-based) simulation, where, for example, all instruction fetches are performed.

The drawback of this approach is that it is not possible to maintain the total separation between the functional and timing information. For example, compiler optimizations can change the source code radically, and different simulation models with different annotations are needed for different processors, compilers and even optimization levels.

Moreover, a serious simulation performance problem could arise from architectures with complex processor condition codes, that can be set by several instructions. On the hardware side, setting condition codes comes at no cost in terms of speed, while the simulation model executes special code for emulating them. Generating the emulation code for all the instructions that alter the condition codes on the processor is definitely a waste of time, since the condition codes are used in branch decisions much less than they are set. There is a known problem in doing this efficiently [5]. While RISC processors often do not use condition codes, several older architectures (e.g., x86) use them extensively, hence an efficient solution for translating them into the C model has been provided.

When the assembler input is parsed, an internal representation of the simulation model is created in the translator. This representation will include all condition code updates. Before the model is output, we perform a data flow analysis on the condition codes and flag as useful only the settings that (conservatively) have a chance to be used by a subsequent conditional branch instruction. When the model is output, only the flagged updates of the condition codes are generated, thus reducing useless condition codes set statements.

5. EXPERIMENTAL RESULTS

Figure 3 shows the block diagram of the Producer-Filter-Consumer design that we used for our experiments. At startup the *Controller* enables the *Producer* to send a frame of an image through Fifo2 to *Filter*. After processing the frame, *Filter* pushes the result to *Consumer* using Fifo3. *Consumer* acts like a sink and sends a received signal to *Controller* via Fifo4. Now *Controller* enables *Producer* through Fifo1 to send a new frame

Table 1: Cycle-accurate (a), source-based (b) and object code-based (c) software performance on PFC test case for none (noopt), moderate (opt), and heavy (oam) compiler optimization.

(a)

task	noopt	opt	oam
producer	1987951	1594577	1594553
controller	421	366	366
consumer	56660	55304	55304

(b)

task	noopt	opt	oam
producer	1982231 (−0.29%)	1982231 (+24.3%)	1982231 (+24.3%)
controller	260 (−38.2%)	260 (−29.0)	260 (−29.0%)
consumer	11035 (−80.5%)	11035 (−80.0%)	11035 (−80.0%)

(c)

task	noopt	opt	oam
producer	1982536 (−0.27%)	1591768 (−0.18%)	1591745 (−0.18%)
controller	1227 (+291%)	1168 (+319%)	1168 (+319%)
consumer	50702 (−10.5%)	49346 (−10.8%)	49346 (−10.8%)

for processing, until there are no more frames to process.

Only *Controller*, *Producer* and *Consumer* are implemented in software on a MIPS3000 processor (the cache model is set to always hit in all cases.) We used three software performance estimation methods:

1. A cycle-accurate simulator for MIPS3000 called TSS, used as a reference (see table 1 (a); total simulation time 9 minutes.)

2. The source-based method (see table 1 (b), total simulation time 30 seconds.) Errors obviously depend on the level of optimization used by the compiler. In the case of *Consumer* they are especially large due to an error in estimating the cost of a function call inside a loop.

3. The object code-based method (see table 1 (c); total simulation time 30 seconds.) The large estimation errors for *Controller* and *Consumer* in this case are due to the different runtime environments, that require a different prologue and epilogue for the routines implementing each block in TSS (a cycle-accurate simulator using memory-based communication) and in the performance simulator (a discrete-event simulator using port-based communication.) They are more relevant than for *Producer* due to the large amount of "useful" (and simulator-independent) computation performed by the latter.

The systematic errors exhibited by *Controller* and *Consumer* in the object code-based case could be reduced by instructing the estimation tool not to analyze prologue and epilogue of top-level functions (managed by the RTOS in one case and by the simulation engine in the other), but to use pre-characterized costs of entry and exit in the real environment (a fixed quantity, that was used by the source-based approach as well.)

6. CONCLUSIONS

System functionality is often implemented in software, yet estimating software performance of embedded multi-tasking reactive systems is still a difficult problem. 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 cycle-accurate ISSs [13].) Reasonably accurate performance estimation is needed in the face of internal pipelines, multiple issue instructions, code and data caches, and memory hierarchies.

In the future, we are planning to apply this technique also to VLIW (Very Long Instruction Word) processors. This type of processors presents an interesting synergy between the compiler design and the hardware design. A VLIW compiler performs static code scheduling without requiring runtime pipeline interlocks, so that the code can be executed without having to take any control decisions at runtime.

7. REFERENCES

[1] *MCORE Reference Manual*. Motorola.

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

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

[4] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques and Tool*. Addison-Wesley, 1986.

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

[6] 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.

[7] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. *Surviving the SOC Revolution – A Guide to Platform-Based Design*. Kluwer Academic, 1999.

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

[9] M.T. Lazarescu, M. Lajolo, J.R. Bammi, E. Harcourt, and L. Lavagno. Compilation-based software performance estimation for system level design. *Proc. Design Automation Conf.*, Jun. 2000.

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

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

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

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