Automatic Generation of Timing Models for Timing Analysis of High-Level Code

Peter Altenbernd
Department of Computer Science
University of Applied Sciences
Darmstadt, Germany
p.altenbernd@fbi.h-da.de

Andreas Ermedahl, Björn Lisper, Jan Gustafsson
School of Innovation, Design, and Engineering
Mälardalen University
Västerås, Sweden
{andreas.ermedahl,bjorn.lisper,jan.gustafsson}@mdh.se

Abstract

Traditional timing analysis is applied only in the late stages of embedded system software development, when the hardware is available and the code is compiled and linked. However, preliminary timing estimates are often needed already in early stages of system development, both for hard and soft real-time systems. If the hardware is not yet fully accessible, or the code is not yet ready to compile or link, then the timing estimation must be done for the source code rather than for the binary. This paper describes how source-level timing models can be derived automatically for given combinations of hardware architecture and compiler. The models are identified from measured execution times for a set of synthetic "training programs" compiled for the hardware platform in question. The models can be used to derive source-level WCET estimates, as well as for estimating the execution times for single program runs. Our experiments indicate that the models can predict the execution times of the final, compiled code with a deviation up to 20%.

1. Introduction

Many safety-critical embedded systems have hard real-time requirements, i.e., failures to meet timing deadlines can have catastrophic consequences. For these, a safe (i.e., never underestimated) Worst-Case Execution Time (WCET) of the software is a key measure.

Often, the hardware is configured in parallel with the software development. For embedded systems in a high-volume market, it is important to choose a suitable hardware configuration (CPU, memory, peripherals, etc.) which is just powerful enough, in order not to have a too costly hardware. Consequently, it is important to assess the worst-case timing of the software to be able to choose a suitable hardware configuration.

However, timing properties are normally verified rather late in the development process, when the code has been compiled and linked, and when the hardware configuration has been determined. If the timing requirements turn out not to be met, then a costly system redesign may be needed. Timing estimates are therefore very useful earlier in the real-time embedded systems development, since they can help getting the dimensioning of the system right (selection of hardware configuration, timing budgets for tasks, etc.). This will reduce the risk that in the end, the system will not meet its timing requirements.

In contrast to the final verification of the timing properties for a safety-critical system, early timing estimates do not have to be safe. Safe bounds derived in early development stages will most likely have to be very pessimistic, due to the lack of detailed information about software and/or hardware. Pessimistic bounds will typically lead to severe overdimensioning of the hardware. It is therefore more important to have a low expected deviation from the final WCET than to have a safe WCET estimate. This is even more true for soft real-time systems.

These early, approximate WCET estimates can be confirmed later in the development cycle. Safe and more exact estimates can be calculated using existing WCET tools as soon as the code has been compiled and linked towards a selected hardware configuration.

In this paper, we present a method to identify a source-level timing model for a given combination of hardware configuration and compiler. The timing model can be used for static source-level WCET analysis as well as estimating performance dynamically with source-level simulators. The approach consists of the following steps:

1) For the source language to be analyzed, a number of "virtual instructions" are selected (such as arithmetic/logic operations, branching, functions calls/return, etc). The set of virtual instructions should define an abstract machine that can execute the source code in a reasonably direct manner.
2) A number of synthetic "training" programs are compiled for the intended platform, and each program is executed. A reasonably timing-accurate simulator can be used, if the hardware is not available. The execution times are recorded.
3) The training programs are executed on an emulator that

can count the number of executions of each virtual instruction. When the training program is executed, the execution count is recorded for each virtual instruction.

4) A linear timing model, with an execution time for each virtual instruction, is automatically identified from the measured execution times and the recorded instruction counts.
5) The resulting timing model can be used to make source-level timing estimates for any program in the selected source language without having access to binaries or target hardware. The timing estimation can be made either by a static WCET analysis or by simulation.

Steps 2) – 4) can be performed once and for all generate a timing model for a given combination of compiler and hardware configuration.

When the timing model is available, it can be used for quick timing estimation while developing the software. The analyzed source code must be complete enough to be translatable into virtual instructions, but calls to yet unknown functions can be present if they are assigned execution times by the user[1]. Complete code snippets can also be analyzed.

If a library of timing models is available for different HW/compiler combinations, then 5) can be performed repeatedly to explore HW/compiler alternatives.

We believe that WCET estimation based on our timing models would fit well into the interactive WCET analysis methodology put forward in [16].

The main contributions of the paper are:

- we have extended state-of-the-art of designing timing models by using dedicated, portable training programs,
- we give insights in what method to use to find the best-fit of the timing model, and
- we have developed novel methods to apply the derived timing models in static timing analysis.

Our experiments indicate that the deviation of execution times predicted by the model from real execution times typically are in the range 0-20%. It should be noted that our WCET estimates are not always safe, due to the approximative nature of the timing model.

The rest of this paper is organised as follows. Section 2 gives an account for related work. In Section 3 we explain the general idea of identifying timing models from measurements. Section 4 describes our approach to source level timing analysis, and gives an overview of our experimental setup. In Section 5 we describe our approach to constructing training program suites for model identification. Section 6 gives an account for the concrete model identification techniques that we have tried out. In Section 7 we describe our experimental setup, experiments, and outcomes, in more detail. Section 8, finally, wraps up.

[1] If the function calls can affect the subsequent program flow, e.g., through side effects, then these side-effects must be captured by value annotations. Most WCET analysis tools support such annotations.

2. Related Work

TimingExplorer [22] is a version of the static WCET analysis tool aiT [8], which is designed for fast design space exploration of different hardware configurations. It sacrifices absolute safety for analysis speed. The analysis uses the binary, and can thus only be applied if the code is ready to compile. The tool relies on hand-crafted hardware timing models.

Another interesting approach for early stage WCET analysis is the integration of timing analysis and modelling/code generation tools (aiT, SCADE, ASCET) [7]. Here, a traditional WCET analysis is done on the binary that is compiled from the generated source code. The tool integration provides support for maintaining mappings between model and code. Since the binary is needed, the models must be in the state that code can be generated from them. Any supporting code that is linked with the generated code must also be compilable. This restricts the portability of the approach to hardware where timing models already exist, and puts restrictions on the use for early phase timing estimation. Similar attempts to do timing analysis for models have been done, targeting statecharts [6], Petri nets [24], and Simulink [19].

There have been approaches to WCET analysis for Java [2, 3, 23], where JVM provides a virtual instruction set. They all rely on manually derived timing models.

Identification of linear timing models has been attempted in the past. Giusto et. al. have investigated the use of model identification for Source-Level Simulation (SLS), which is used for performance evaluation of embedded software [11]. A more recent, similar approach is reported in [17]. Franke [9] has used the approach on machine code level, to build binary level simulators where approximate execution times are calculated quickly from instruction counts. Franke's models also include data such as cache miss ratio, which is recorded by the simulator. What sets our work apart from these contributions is the systematic design of training programs, the use of other methods than the least-squares method for model identification, and the use of the derived models in static timing analysis.

Lisper and Santos have developed a new regression method, based on end-to-end measurements of programs, where the resulting timing model is guaranteed to not underestimate any observed execution times [21]. The technique works for binaries, identifying execution times for basic blocks in specific programs, which is different from our work. In [1], model identification via trace parsing and machine learning is used to derive parametric loop-bounds. The method is similar to ours but the aims are totally different. In [13], the approach that has been pursued here is outlined.

Finally, an approach similar to ours has been used to fit energy consumption models for low-level code [20].

3. Identification of Linear Timing Models

Assume an abstract machine with virtual instructions $i_1, \ldots, i_n$, modelling a source language. We assume that a program in the source language is emulated by the abstract machine, so that the execution of a program in the source language corresponds to a trace of virtual instructions. For each virtual instruction i_k, the trace then contains c_k occurrences of i_k (the *execution count* of i_k). A *linear timing model* for the abstract machine computes the execution time T for a trace as

$$T = w_0 + \sum_{k=1}^{n} w_k c_k \quad (1)$$

Here, w_0 is a constant startup time, and w_k, $k = 1, \ldots, n$ are constant execution times for the respective virtual instructions. If we assume that i_1 is a virtual "startup" instruction, which occurs exactly once in each trace, then (1) can be simplified to

$$T = \sum_{k=1}^{n} w_k c_k \quad (2)$$

A linear timing model can predict the real execution time for the compiled binary more or less well. Good linear timing models can be expected to exist for code compiled with non-optimizing compilers, and executing on simple hardware without features such as pipelining or caches. For more complex hardware architectures, with widely varying instruction execution times, linear models will be less accurate. The same is true if an optimizing compiler makes heavy changes in the code structure. However, approximate models may be useful for early timing estimation.

A common situation is that we want to find a "best" model, i.e., an assignment of values to the instruction execution times w_i that minimizes the overall deviation of the predicted execution times from the real ones for a number of observations. Each observation j consists of a measured execution time t_j for the compiled binary, and an array $(c_{j1}, \ldots, c_{jn})$ of execution counts for the emulated source code executed with the same inputs as the compiled binary. Assume we have made m observations. The model then predicts an array $C\vec{w}$ of execution times, where C is an $m \times n$-matrix whose rows are the observed arrays of execution counts, and $\vec{w} = (w_1, \ldots, w_n)$ is an array of virtual instruction execution times. Let $\vec{t} = (t_1, \ldots, t_m)$ be the array of measured execution times for the different observations. Finding a "best" model then amounts to finding a $\vec{w}$ that minimizes the overall deviation of $C\vec{w}$ from $\vec{t}$.

There are different ways to define the overall deviation. The *Euclidean distance* is often used for this purpose:

$$\|\vec{x} - \vec{y}\|_2 = \sqrt{\sum_{j=1}^{m} (x_j - y_j)^2}$$

The well-known *least-squares method* (LSQ) [10] will find a $\vec{w}$ that minimizes $\|C\vec{w} - \vec{t}\|_2$. It is commonly used for linear model identification, and has been used to identify linear timing models before [9, 11, 17].

In order to obtain a good linear timing model, the set of observations must contain enough information to allow for an accurate identification of each w_i. If there are linear dependencies in the observations between the execution counts for different virtual instructions (i.e., if there are sets of linearly dependent column vectors in C), then LSQ can yield multiple solutions. This can result in arbitrarily bad execution time predictions for other executions for which the linear dependencies are not obeyed. Linearly independent but highly correlated virtual instruction counts can also yield models with poor predictions. In order to avoid this, the observations must contain enough variation in the execution counts such that the linear dependencies go away. This amounts to having a good enough set of training programs, and test data. However, some linear dependencies might be inherent in the trace structure: for instance, virtual *CALL* and *RETURN* instructions can be expected to have the same execution counts for any observation. Such a set of linearly dependent instructions can always be replaced by a smaller set of virtual "superinstructions" for which unique execution times can be identified. In the example, *CALL* and *RETURN* can be replaced by a single superinstruction carrying the sum of their execution times.

Even if linear dependencies are not present, it is not certain that LSQ will yield a model that always predicts execution times well for executions outside the set of observations used to identify the model. In practice, one might want to use knowledge about what should be "reasonable" instruction execution times to constrain the solution. This motivates the use of other, more general search or optimization methods for the model identification. Such methods also make it possible to minimize some other distance measure than the Euclidean distance.

4. Early-Timing Analysis Approach

An overview of our approach, and experimental setup, is shown in Fig. 1. Once the training programs have been designed (see Section 5) they are compiled and executed on the target hardware, or a simulator for it. For convenience we have used the SimpleScalar simulator [4] for this purpose: SimpleScalar can be configured to simulate architectures with a variety of different features, and it records the number of cycles needed to execute the program on the virtual hardware. These values form the vector $\vec{t}$.

In the next step, following [13], the training programs are translated into the intermediate format ALF [15], which provides the virtual instruction set. The C-2-ALF translator that we have used produces ALF code that is a faithful image of the C code, with preserved program

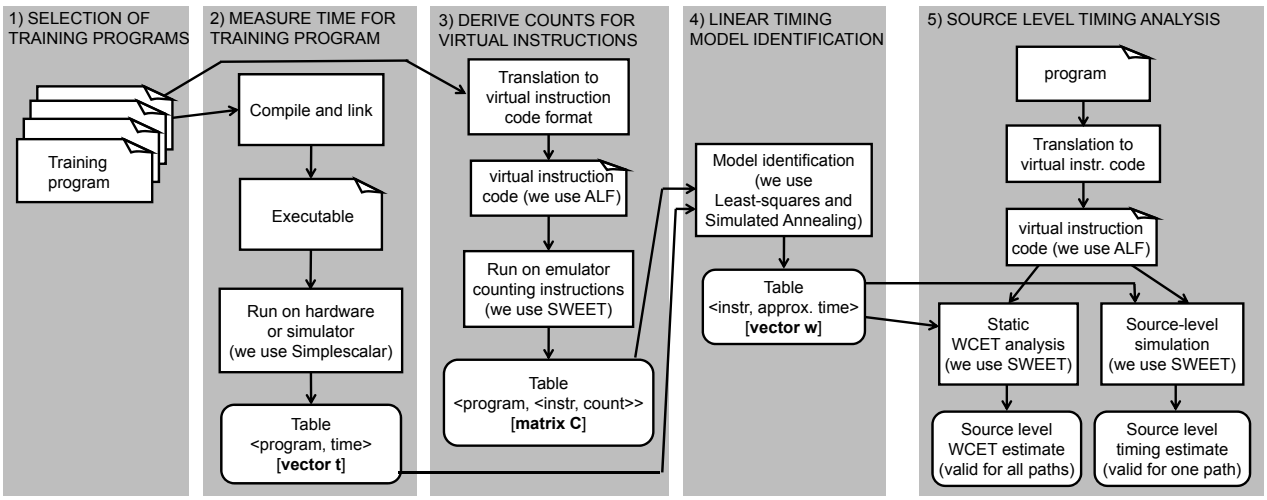

Figure 1. Early-timing analysis approach.

flows[2]. A modified version of the WCET analysis tool SWEET [5] interprets the ALF code, and records the execution counts for the ALF instructions. These execution counts form the matrix C.

Then the model is identified, i.e., the vector $\vec{w}$ is determined. We have tried the least-squares method, and simulated annealing for this purpose, see Section 6.

The final step is to use the model for timing analysis. This can be done either through simulation, or by a static timing analysis. We have used SWEET to do both simulation, and an approximate static WCET analysis on source level using the timing models. For the simulation, we extended the interpretation mode of SWEET to provide timing estimates using ALF timing models. See further Section 7.

As mentioned in Section 1, steps 1) – 4) can be done once and for all for a given HW/compiler configuration. 5) can then be performed repeatedly by the user to estimate WCET's on source code level.

5. Construction of Training Programs

The selection of training programs is of utmost importance to the timing model identification. It has been proposed to use a mix of codes from the intended application domain [11]. A problem with this approach is that it is not ensured that the mix of codes will provide enough independence in the execution counts of different virtual instructions to avoid problems with linear dependencies, or high correlation between instruction counts. We therefore advocate an approach with synthetic training program suites,which allows more control over the virtual instruction traces. The program suites can quite easily be designed to avoid the problem of linearly dependent, or highly correlated execution counts. We have designed training program suites for two scenarios: simple architectures without caches etc., where we can expect close to constant instruction execution times, and advanced architectures with caches, pipelines, parallel functional units, etc.

[2] `www.complang.tuwien.ac.at/gergo/melmac/`

5.1. Training Programs for Simple Architectures

For simple architectures, where the execution times of the instructions have low variability, the execution times of virtual instructions can be estimated well regardless of which other instructions are executed. Ideally then, one would construct a single training program for each virtual instruction executing that instruction once. Executing each training program once would then yield an execution count matrix C which is the identity matrix, and the vector $\vec{w}$ would equal the vector $\vec{t}$ of observed execution times. Of course, such a program suite is not possible since typically some instructions are needed in order to execute others. But the idea can be used to construct a training program suite where single instructions are isolated as far as possible. Thus, our suite for simple architectures is constructed like this:

- The first program is the "empty" program. Execution yields the startup time for a program (i.e., , the time for a virtual *RUN_PROG* statement):

  ```
  int main() { }
  ```

  This program is put first in the suite, as every emulation of virtual instructions for a source program will execute *RUN_PROG*.

- Any nonempty C program must contain an assignment. Such a program must execute at least one virtual *STORE* instruction. Thus, the next program executes exactly one *STORE*:

  ```
  int main() { int j = 17; }
  ```

- Similarly, a third program executes a *LOAD* instruction in addition to a *RUN_PROG* and a *STORE*. The program suite then continues with a series of simple programs, executing each remaining instruction in turn, until the full set of instructions is covered. An example is the training program for the *INT_MULT* instruction:

```
int main() { int j = 42; j = j*3; }
```

However, one dependency cannot be eliminated by this regime. The number of executed *RETURN* instructions will always equal the number of function calls. But if these instructions are replaced by a superinstruction carrying their added execution times, then this approach will yield a lower-triangular execution count matrix C, with nonzero diagonal entries. Such a matrix will never have any linear dependencies between the column vectors. C will furthermore be a quadratic matrix, since there is exactly one program execution per instruction: then, the absence of linear dependencies implies that C is invertible, and the linear system $C\vec{w} = \vec{t}$ can be solved directly to yield $\vec{w}$ such that $\|C\vec{w} - \vec{t}\|_2 = 0$.

5.2. Training Programs for Advanced Architectures

For more advanced architectures, features such as caches and pipelines will cause instructions to have highly context-dependent execution times. This means that the "real" instructions must be executed in a variety of contexts when identifying the timing model, in order to have the model capturing the influence of these contexts on the execution time. In particular, longer instruction sequences must be executed in order to capture cache and pipelining effects. One way to accomplish this is to introduce loops in the code. Our "advanced architecture" test suite builds on the suite for simple architectures, and extends the programs with loops executing the instruction under test a number of times. This simple extension will not capture more complex timing effects involving several different instructions, but can still give reasonably good results as we have verified in Section 7.1.

However, when introducing loops then some instructions will invariably be needed: a *STORE* to initialize the loop variable, some arithmeric operation to increment/decrement the loop counter, some test instruction to decide the exit condition, a *JUMP* to return to the entry point of the loop. Therefore, we cannot obtain a lower-triangular execution count matrix any more. But the execution counts can be made linearly independent by introducing several loops executing different arithmetic operations to increment/decrement the loop counter, and different test instruction to break the loop. A third part of the code executes the loop body outside any loop, to break the possible linear dependencies to instructions (like *JUMP*) that appear in all loops. If the training program is executed a number of times, with the different loop bounds set in a linearly independent fashion, then the linear dependencies between any instructions will be broken, and in addition the correlation can be brought down.

An example is the training program for *INT_MULT*, which consists of two independent loops, and a section with straight-line code:

```
int main() {
  int max1 = ...;
  int max2 = ...;
  int i,j;
  for (i=1; i <= max1; i++) {
    j = i; j = j*3;
  }
  for (i=max2; i > 0; i--) {
    j = i; j = j*3;
  }
  j = i; j = j*3;
}
```

If only the first loop was present, then *INT_MULT* would always be executed as many times as the *ADD* that increments `i`, and the test operation that compares `i` with `max1`. This would create a linear dependency between these execution counts. The second loop uses a *SUB* to decrement the loop counter, and a different test operation.

Thus, a number of executions can be made where `max1` and `max2` are varied in such a way that the resulting execution counts for the involved virtual instructions are linearly independent. However, both loops will still have a single *JUMP* instruction each: thus, the execution counts for *JUMP* and *INT_MULT* will still be the same no matter what `max1` and `max2` are. To break this linear dependency, the third appearance of the loop body is added.

In our automated approach, in which the training programs are generated, the constants `max1` and `max2` can be varied (see Section 7.1).

6. Model Identification

We have tried out some different approaches to the problem of choosing $\vec{w}$ such that the resulting source level timing model predicts the execution time well for the compiled binary, running on the chosen target platform. We have LSQ (see Section 3) for this purpose, which always gives the best fit for the set of training program runs as defined by the Euclidean distance. However, this method will blindly select real-valued weights to minimize this distance, and so it can yield models that do not predict the execution time well for programs outside the training set. For instance, it may yield cycle counts for instructions that are negative, and indeed this sometimes happened in our experiments. It is clear that such execution times are unrealistic in practice, and may yield very poor predictions for programs that execute the instruction in question very frequently relative to other instructions.

Thus, we have also used a more general search method that allows more freedom in specifying constraints, and

objective function. Among the many approaches of this kind we opted for simulated annealing [18] (SA): it is relatively simple to implement, and compared to simple greedy algorithms it is capable of overcoming local optima.

By analogy with a physical process of cooling, each step of the SA algorithm replaces the current solution by a random solution from the neigbourhood. If better, the new one is accepted. If not, it still might be accepted with a probability that depends both on the difference between the subsequent objective function values, and on a global parameter T (the "temperature"). As the temperature is decreased during the process, jumps leaving the local solution space will become less and less likely, so eventually the result will stabilise.

Adapting SA to minimize $\|C\vec{w} - \vec{t}\|_2$ is easy, and it is done according to the following:

- All elements in $\vec{w}$ are initialised to zero.
- For producing a solution in the neigbourhood of $\vec{w}$, its elements will be randomly incremented or decremented by one, or kept as is (while upholding any imposed constraints on the solution).

The rest remains as in the original SA algorithm. However, SA is very sensitive to its steering parameters, like the temperature, and how it is reduced. Therefore we have run SA several times, with varying parameters, and kept the best result.

7. Experiments

We have evaluated the precision of the identified models, as well as the influence of the training program suite on the result. We have used the two sets of training programs from Section 5.2 for this purpose, and we have tried both LSQ and SA. The models were then evaluated using a distinct set of programs, consisting of fifteen programs from the Mälardalen WCET Benchmark Suite [14], see Table 1. Table 1 gives some basic data about the programs, including lines of C code (**#LC**), the number of functions (**#F**), loops (**#L**), and conditional statements (**#C**).

We first compared predicted vs. real running time for each benchmark and model, running each benchmark with its specified input (all these benchmarks have their hard-coded inputs). This gave an estimation of how well the derived timing models predict real running times. We then removed the hard-coded inputs for some selected benchmarks, turning them into programs having different paths through the code for different inputs. For each selected benchmark, ranges were defined for the possible input values. To evaluate the precision of static timing analysis based on the timing models, we finally performed a static BCET/WCET analysis for these benchmarks taking their input ranges into account. For each benchmark, these estimates were compared with the best/worst running times obtained by an exhaustive search over the possible inputs.

We have used SWEET both for the single runs and the static analysis. SWEET has a "single-path mode" that can be used to emulate ALF code. We have extended this mode to use ALF timing models, effectively turning SWEET into a source-level simulator estimating execution times. We have also extended SWEET's static analysis to perform BCET analysis in addition to WCET analysis, which is straightforward for simple timing models with constant execution times for basic blocks like the ALF timing models considered here.

We have used the SimpleScalar simulator [4] as target architecture. Both the training programs, and the benchmarks used for the evaluation were compiled using `sslittle-na-sstrix-gcc` with no optimizations, for `sim-outorder` executed with its standard configuration. With this configuration, `sim-outorder` simulates a processor with out-of-order issues of instructions, main memory latency 10 cycles for first access and 2 cycles for next accesses, memory access bus width 64 bytes, 1KB L1 instruction cache (1 cycle, LRU), no data cache, no L2 cache, no TLB's, 1 integer ALU, 1 floating point ALU, and fetch width 4 instructions. Branch prediction is 2-level with 1 entry in the L1-table, 4 entries in the L2-table and history size of 2. Our benchmarks only use integers operations.

Since none of the selected benchmark programs use any floating-point instructions the translated ALF programs use only 31 different ALF instructions, which formed the virtual instruction set for the experiment. These instructions include program flow control instructions, *LOAD*/*STORE*, and arithmetic/logical instructions excluding floating-point arithmetics.

The experiments were run on a PC with a quad-core 2.53 GHz Intel processor, equipped with 4 GB memory, running Linux with kernel 2.6.35-997-generic.

7.1. Training Programs

We first used the "simple" training program suite from Section 5.1. For the set of benchmark programs in Table 1, we obtained an average deviation of 29%. This rather poor precision shows that this suite is not well suited to identify models for architectures like `sim-outorder`.

For the "advanced architecture" suite we tried to vary the number of iterations of the loops, to see whether this would have an influence on the precision of the predicted execution times. Since architectural features like caches, and branch predictors tend to yield shorter instruction execution times within loops, it seemed likely that this could influence the identified model and its resulting precision. To estimate this influence we used the program suite instantiation "small" (loops iterating 7 - 17 times), "medium" extending "small" with instances of the programs iterating up to 29 times, and "big" extending "medium" with instances iterating up to 61 times. We also tried adding the "simple" training program suite to see what influence this would have on the precision. This gave a total of six test cases with different variations of the

Program	Description	#LC	#F	#L	#C
bs	Binary search in an array of 15 integer elements.	114	2	1	3
cover	Program with many paths (using loops and switches).	640	4	3	6
edn	Finite Impulse Response (FIR) filter calculations.	285	9	12	12
esab_mod	Loop with highly context-sensitive execution.	3064	11	1	292
fdct	Fast Discrete Cosine Transform.	239	2	2	2
fibcall	Iterative Fibonacci, used to calculate fib(30).	72	2	1	2
fir	Finite impulse response filter (signal processing).	276	2	2	4
inssort10	Insertion sort on a reversed array of size 10.	92	1	2	2
inssort15	Insertion sort on a reversed array of size 15.	92	1	2	2
inssort20	Insertion sort on a reversed array of size 20.	92	1	2	2
inssort30	Insertion sort on a reversed array of size 30.	92	1	2	2
jcomplex	Nested loop program.	64	2	2	4
loop3	Loops with context-sensitive execution behaviour	76	1	150	150
ns	Search in a multi-dimensional array.	535	2	4	5
nsichneu	Simulates an extended Petri net (many paths).	4253	1	1	253

Table 1. Benchmark programs

Training suite	LSQ	LSQ rounded	SA	SA $\geq$ 0	0 $\leq$ SA $\leq$ 2$\times$
small	39%	34%	10%	10%	10%
medium	50%	45%	14%	12%	12%
big	64%	63%	16%	13%	13%
small + simple	18%	17%	15%	10%	20%
medium + simple	19%	15%	16%	10%	10%
big + simple	17%	14%	16%	10%	10%

LSQ: standard least squares method, **LSQ rounded**: LSQ rounded to closest integer
SA: Simulated Annealing with no constraints on the solution
SA $\geq$ 0: SA restricted to nonnegative instruction times, **0 $\leq$ SA $\leq$ 2$\times$**: SA additionally restricted from above

Table 2. Average deviation of predicted vs. real execution times for benchmarks with different model identification methods

training program suite.

7.2. Model Identification Method

We tried different variations of LSQ and SA. For LSQ we tried both the direct solution, where virtual instruction execution times can be non-integers, and with instruction execution times rounded to the closest integer. In both cases, no further restrictions were made on the instruction execution times: therefore, e.g., negative execution times were possible and would indeed appear sometimes.

For SA, we used the Euclidean deviation $\|C\vec{w} - \vec{t}\|_2$ as objective function. We used three variations. In the first, SA was run without any constraints on the final solution. For the second, we imposed the constraint that all virtual instruction execution times be nonnegative. The rationale for this was to see whether this "sanity" constraint would yield better predictions overall.

In the third variation, we restricted the search space size for SA by imposing also an upper limit on the virtual instruction execution times. This is a way to curb potentially long search times for SA, and we wanted to see whether this would affect the precision in the identified model adversely. In order to set the upper limit, we first made a rough estimate using the "simple" training program suite. As this suite has exactly one training program per virtual instruction, its matrix C is invertible and the equation system $C\vec{w} = \vec{t}$ can be solved exactly by standard methods, in very short time, to yield $\vec{w}_{simple}$ forming the rough estimate. We then restricted the SA solution $\vec{w}$ from above by $2\vec{w}_{simple}$.

Since SA was initialized with an integer vector, using integer steps in the search, it would always return an integer vector as solution.

All the variations of LSQ and SA were tested with all the six different training program suite combinations. The results are shown in Table 2. In all cases, the relative average deviation of predicted running times from measured running times are given in percent.

SA clearly outperforms LSQ in all examples. LSQ performs poorly for the pure small/medium/big training suites, but the results are significantly improved in all cases if the simple training suite is added. Surprisingly, rounding the LSQ solution to integers will not have an adverse effect: rather, on the contrary. The consistently best results are given by SA restricted to nonnegative values. Restricting the search range from above did not have any adverse effect at all, except for the small + simple training suite. The restriction resulted in faster convergence for SA in all cases. In our experiments the time for performing a full model identification never exceeded a couple of min-

Program	Model	Measured	Diff	Rel. diff
bs	274	317	-43	-13.6%
cover	3515	8388	-4873	-58.1%
edn	244189	232561	11628	5.0%
esab_mod	698848	699934	-1086	-0.2%
fdct	9250	11294	-2044	-18.1%
fibcall	788	901	-113	-12.5%
fir	6973	8468	-1495	-17.7%
inssort10	3674	3529	145	4.1%
inssort15	549	579	-30	-5.2%
inssort20	729	759	-30	-4.0%
inssort30	1089	1119	-30	-2.7%
jcomplex	671	673	-2	-0.3%
loop3	11999	13371	-1372	-10.3%
ns	31897	33718	-1821	-5.4%
nsichneu	19744	18545	1199	6.5%

Table 3. Predicted vs. measured times for single benchmark program runs

Program	Model	Measured	Diff	Rel. diff
bs	130	184	-54	-29.3%
cover	1837	3605	-1768	-49.0%
edn	140136	119291	20845	17.5%
esab_mod	368743	408076	-39333	-9.6%
fdct	4998	3940	1058	26.9%
fibcall	283	377	-94	-24.9%
fir	3923	4035	-112	-2.8%
inssort10	2094	1678	416	24.8%
inssort15	284	303	-19	-6.3%
inssort20	379	395	-16	-4.0%
inssort30	569	568	1	0.2%
jcomplex	282	307	-25	-8.1%
loop3	5017	6290	-1273	-20.2%
ns	18758	18725	33	0.2%
nsichneu	10969	10129	840	8.3%

Table 4. Predicted vs. measured times for single benchmark program runs, advanced architecture

utes, for any combination of method and training suite, so from that perspective faster convergence is not essential. We do believe, however, that it can be a useful technique if larger training suites are used since the solution space, and thus the potential search time, grows quickly with each new training case.

7.3. Results

Table 3 shows the deviations between measured and predicted running time for the individual benchmark programs, for the best model obtained by SA ≥ 0. All programs have deviations from close to zero up to about 20% except `cover`, which is an extreme outlier with more than 50% underestimation. The reason for this underestimation is that the virtual instruction set used in our experiments has a single instruction for `switch`, which therefore is assigned a constant cost. However, in reality the cost also depends on the number of executed `case` statements. `cover` is an artificial program with very many `case` statements, executing over 50 `case` per `switch` on average. Since the execution times for the `case` statements are not accounted for in the model, the heavy underestimation results. Clearly, a redesign of the virtual instruction set to model the `case` statements better would allow for better timing models of this program.

The approximate source-level timing analysis was evaluated using a subset of benchmark codes. These programs were selected since they are all multipath programs, where the execution time varies with the inputs, and since their worst- and best-case inputs are known. Thus, the real BCET and WCET could be approximated by running SimpleScalar with the proper inputs. (We do not know the worst- and best case initial hardware states, so we cannot guarantee that our values are the real BCET and WCET, but we expect them to be close.)

The results of the analysis are shown in Table 5, together with the corresponding measured BCET and WCET recorded from SimpleScalar. The timing analysis yields reasonable BCET and WCET estimates, with precision comparable to the precision for predicting single execution times. The estimates are not always safe, due to the approximative nature of the timing model. (In table 5, a positive Diff means that the corresponding BCET/WCET estimation is safe, i.e., $BCET_e \leq BCET_m$ and $WCET_e \geq WCET_m$.)

7.4. Advanced architecture

Our results were obtained for the default hardware configuration of `sim-outorder`, where it simulates a relatively simple processor with only a small instruction cache and no parallel integer functional units. To see whether our method for model identification would yield reasonable timing models also for a more advanced architecture, we configured `sim-outorder` to simulate a processor with the following characteristics: out-of-order issues of instructions, main memory latency 18 cycles for first access and 2 cycles for next accesses, 8KB L1 data and instruction caches, respectively (1 cycle), 256KB L2 data and instruction cache (6 cycles), all caches LRU, no TLB, 2 integer ALU's, 2 floating-point ALU's, fetch decode, issue, and commit width all 4 instructions, perfect branch prediction.

We then re-ran the model identification for this hardware configuration. Again, the best models were obtained by SA ≥ 0. We evaluated the precision of the timing model using single benchmark program runs, corresponding to Table 3. The deviations are in the range 0-30%, rather than 0-20% for the simpler architecture, and the average deviation is 15% rather than 10%. The results per benchmark run are shown in Table 4.

7.5. Discussion

The best combination of method and training suite is medium/big + simple and SA with nonnegative virtual instruction execution times. With this combination, we obtained an average relative deviation of 10% for the single runs of the selected benchmarks with results in the range 0-20%. For the BCET/WCET estimates, we obtained a deviation from the real WCET in the range 0-25%. We find these results surprisingly good, given the simplicity of the model. For the advanced architecture the deviations

Program	BCETm	BCETe	Diff	Rel. Diff	WCETm	WCETe	Diff	Rel. Diff
jcomplex	78	65	13	16.7%	708	715	7	1.0%
inssort10	465	487	-22	-4.7%	3529	3682	153	4.3%
fibcall	79	71	8	10.1%	2861	2546	-315	-11.0%
edn	231825	243415	-11590	-5.0%	232561	244197	11636	5.0%
bs	154	117	37	24.0%	315	283	-32	-10.1%

BCETm/WCETm: measured BCET/WCET, **BCETe/WCETe**: estimated BCET/WCET

Table 5. BCET/WCET analysis results

are larger, but still the model makes reasonable predictions.

Constraining instruction execution times to be nonnegative is clearly beneficial for the precision of the models. An advantage with search methods like SA is that they allow a large freedom in choosing constraints and objective functions. In our experiments, the time to identify a model by SA was not a big issue. Therefore, we believe that search methods like SA is a good choice for source level timing model identification.

Our training programs are small, with tight loops and a high degree of data locality. This will of course yield a high cache hit ratio. The models obtained can be expected to give good predictions for code with similar cache hit ratio. Indeed many of the benchmarks, although not all, are small and have tight loops. It is, however, interesting to note that we obtained models with better precision when adding the "simple" training programs, which terminate very quickly without looping and thus can be expected to have a lower cache hit ratio. In general, the structure of the training programs should match the structure of the expected mix of application programs – if the application domain programs contain, say, large loops with little cache reuse, then the training programs should be designed in a similar fashion. If they are synthetic, they can still be designed to yield execution count matrices with good properties such as linearly independent column vectors having a low degree of correlation.

We have obtained our results for unoptimized code. In many safety-critical applications compiler optimizations are disallowed: then, our method should work well. Optimizations can however be expected to lead to less accurate source-level timing models, since the structure of the binary will be less similar to the structure of the source. We believe that the situation can be somewhat improved by designing the training programs to allow "typical" optimization performed by compilers. This requires more complex training programs than the ones described here. Again, synthetic training programs can be designed to yield execution count matrices with good properties.

Finally, it is worth noting that a static source-level timing analysis, using the timing models, usually can be done with a high level of automation since source-level program flow analysis typically is easier to do, with high precision, on source-level than for binary code. In our experiments all the statically computed BCET and WCET estimates in Table 5 were found by a fully automatic analysis, without the need for any manual annotations constraining the program flow.

8. Conclusions, and Further Research

We have shown that source-level timing analysis can be done, with good accuracy, using an approach where a source-level timing model is identified from executions of training programs designed to allow fast identification of accurate models. A static source-level timing analysis, using the timing model, can compute approximate BCET and WCET estimates with a high degree of automation. These estimates can be used in early stage system development for purposes like time budgeting, where they can help making reasonable time budgets that likely can be met. Using timing models for different combinations of compilers and hardware, the method allows for rapid design space exploration to choose the best combination for the application at hand.

We found that a general search method, like Simulated Annealing, is suitable for the model identification due to its robustness and the ability to impose different constraints on the resulting model. It should be noted, though, that algorithms for the Least-Squares method with linear constraints exist [12] and could have been used to impose the nonnegativity constraint in our experiments. How to apply such methods to source-level timing analysis remains a topic for further research.

Other topics for further research are how to build source-level timing models that are more precise, as well as how to find models that give reasonably good results in situations with a more complex mix of programs, and optimizing compilers. In [25], an approach to creating timing models for optimized code using intermediate code was developed, but this approach requires that the code compiles and is thus less applicable to early timing analysis. A final issue for further investigation is whether our approach can be taken further to automatically identify timing models on model level, to be used in model-based development.

9. Acknowledgments

This work was partially supported by VINNOVA through the TIMMO-2-USE project, by VINNOVA and Artemis through the CHESS project, and from the European Commission through the Lifelong Learning

ERASMUS-Programme. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

References

[1] M. Bartlett, I. Bate, and D. Kazakov. Guaranteed loop bound identification from program traces for WCET. In *Proc. 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*, pages 287–294, San Francisco, CA, Apr. 2009. IEEE Computer Society.

[2] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, pages 39–48, 2000.

[3] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis using Java Byte Code. In *Proc. 12th Euromicro Conference of Real-Time Systems, (ECRTS'00)*, pages 81–88, Stockholm, 2000.

[4] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3), 1997.

[5] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.

[6] E. Erpenbach and P. Altenbernd. Worst-case execution times and schedulability analysis of statecharts models. In *Proc. 11th Euromicro Conference of Real-Time Systems*, pages 70–77, 1999.

[7] C. Ferdinand and R. Heckmann. Worst-case execution time – a tool provider's perspective. In *Proc. 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC2008)*, pages 340–345, Orlando, Fl, USA, May 2008.

[8] C. Ferdinand, R. Heckmann, and B. Franzen. Static memory and timing analysis of embedded systems code. In *3rd European Symposium on Verification and Validation of Software Systems (VVSS'07), Eindhoven, The Netherlands*, number 07-04 in TUE Computer Science Reports, pages 153–163, Mar. 2007.

[9] B. Franke. Fast cycle-approximate instruction set simulation. In H. Falk, editor, *Proc. 11th International Workshop on Software and Compilers for Embedded Systems (SCOPES'08)*, pages 69–78, Munich, Mar. 2008.

[10] C. F. Gauss. *Theoria motus corporum coelestium: in sectionibus conicis solem ambientium / auctore Carolo Friderico Gauss*. Sumtibus F. Perthes et I.H. Besser, Hamburg, 1809.

[11] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proc. Conference on Design, Automation and Test in Europe (DAC 2001)*, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[12] G. H. Golub and M. A. Sanders. Linear least squares and quadratic programming. Technical Report CS 134, Computer Science Department, Stanford University, May 1969.

[13] J. Gustafsson, P. Altenbernd, A. Ermedahl, and B. Lisper. Approximate worst-case execution time analysis for early stage embedded systems development. In S. Lee and P. Narasimhan, editors, *Proc. Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, pages 308–319, Newport Beach, CA, USA, Nov. 2009. Springer-Verlag.

[14] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In B. Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 137–147, Brussels, Belgium, July 2010. OCG.

[15] J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, and L. Källberg. ALF – a language for WCET flow analysis. In N. Holsti, editor, *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009)*, pages 1–11, Dublin, Ireland, June 2009. OCG.

[16] T. Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California at Irvine, 2009.

[17] A. Holzer, V. Januzaj, S. Kugele, and M. Tautschnig. Timely time estimates. In T. Margaria and B. Steffen, editors, *Proc. 4th International Symposium on Leveraging Applications of Formal Methods (ISOLA'10), Part I*, volume 6415 of *Lecture Notes in Comput. Sci.*, pages 33–46, Heraclion, Crete, Oct. 2010. Springer-Verlag.

[18] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34:975–986, 1984. 10.1007/BF01009452.

[19] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully automatic worst-case execution time analysis for Matlab/Simulink models. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02)*, Washington, DC, USA, 2002.

[20] S. Lee, A. Ermedahl, S.-L. Min, and N. Chang. An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processor. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, pages 1–10, 2001.

[21] B. Lisper and M. Santos. Model identification for WCET analysis. In *Proc. 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*, pages 55–64, San Francisco, CA, Apr. 2009. IEEE Computer Society.

[22] S. Nenova and D. Kästner. Source level worst case timing estimation and architecture exploration in early design phases. In N. Holsti, editor, *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009)*, pages 12–22, Dublin, Ireland, June 2009. OCG.

[23] P. Persson and G. Hedin. Interactive execution time predictions using reference attributed grammars. In *Proc. of the 2:nd Workshop on Attribute Grammars and their Applications (WAGA'99), Netherlands*, pages 173–184, Aug 1998.

[24] F. Stappert. *From Low-Level to Model-Based and Constructive Worst-Case Execution Time Analysis*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2004. C-LAB Publication, Vol. 17, Shaker Verlag, ISBN 3-8322-2637-0.

[25] Z. Wang and A. Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. In *Proc. 46th Design Automation Conference (DAC 09)*, San Francisco, USA, July 2009.