Evaluation of an Additive WCET Model for Software Components

Marcelo Santos, Björn Lisper[1]

[1] School of Innovation, Design and Engineering, Mälardalen University, Sweden.

{marcelo.santos,bjorn.lisper}@mdh.se

Abstract. *The use of component technology in embedded systems brings new challenges to this domain. One important issue is how to derive properties of the system when we know the properties of the system's components. When the system is real-time, it is useful to find out how component composition will affect the execution time of tasks made out of components. In this work, we use a statistical design to examine and rank different hardware features with respect to their impact on the execution time at component composition. Our results indicate, for example, that main memory latency and the block size of the L2 cache have the greatest effect, while memory bandwidth has the least effect.*

1. Introduction

Timing Analysis giving program timing characteristics is of fundamental importance to the successful design and execution of real-time systems. One key timing measure is the *worst-case execution time* (WCET) of a program. A WCET analysis is based on the assumption that the program is executed in isolation from beginning to end without interruption, and has the goal of finding an upper bound to the worst possible execution time of the program. Reliable WCET estimates are a key component when designing and verifying real-time systems, and are needed in hard real-time systems development to perform, for example, scheduling and schedulability analysis [6].

The WCET is often estimated through measurements, and so is not reliable in general [16], as it is hard to find the input parameters that cause the worst execution time. An alternative is *static WCET analysis*, which determines a timing bound from mathematical models of the software and hardware. If the models are correct, then the analysis will derive a safe timing bound. For simple architectures, the analysis gives a good result. Nowadays, the trend in embedded systems is to use more complex processor architectures with more advanced features that enhance the overall performance of the processor, providing more functionalities to the user, and this puts hard limits on static analysis. The timing effects of pipelines are mostly local, and can be handled quite well, as in Engblom [4]. Cache memories [5] and branch predictors [2] are harder to analyze, since they yield a global, history-dependent timing model, and static analyses for them attempts to estimate their states, as in Bodin and Puaut [2]. Instruction-level parallelism, often through a superscalar architecture, can be even harder to analyse with respect to timing properties due to their dynamic instruction scheduling, but attempts have been made, as in Lim et al. [11].

Component Based Embedded Systems is an application of component technology to embedded systems development, which conceptionally works by dividing the system into smaller, manageable pieces. This has the potential to help build better and more reliable

software, but to achieve this goal with this methodology is a very difficult task due to the ever increasing size, complexity and requirements of modern software systems. We can mitigate this if we increase our ability to combine existing pieces of software to produce new applications (see for example Wiederhold et al. [15]), mainly if they can be selected and assembled in different combinations to satisfy specific requirements or properties. Better yet if such properties can be inferred and composed like the components. Besides these, there are several advantages of using component–based development in the embedded systems domain. Reduction of complexity and fast development for large systems are definitely big issues. If the analysis of the components is also *compositional*, we can easily derive the analysis of the new system by simply composing the analysis of its components. This is particularly useful if the source code is missing or if copyright issues prevent the reverse-engineering of binary code for analysis. In some cases even an approximate model for the composition can be quite useful as, for example, in the early design phase, when the execution environment is not well defined. In component models in general, the composition is not a well defined entity, as noted in the survey by Lau and Wang [10], and hardly say anything about composing functional and non-functional properties (these might be hard to quantify, like quality of service, and are an active area of research). Some models even lack important properties for their domain of application, like the Autosar, that lacks aspects of timing [13]. But a few exceptions exist, like the Koala component model (see van Ommering et al. [14]), where it is possible to infer the total memory consumption based on the consumption of the components.

In order to be able to successfully apply a compositional analysis to component-based technology, we need to provide, between other things:

- some means of deriving the analysis for a component in isolation; by doing this, the component can be delivered (or sold) with the analysis (that is, we can *reuse* the analysis as well);
- and some way of composing the analysis in new composites (component assemblies). The composition can be a simple arithmetic expression or a complex parametric expression derived from some abstract analysis.

Reuse of Timing Analysis is one of the goals of applying component technology to embedded systems development, besides reusing the components themselves. Several types of analysis are of interest in this domain, and for real-time systems the WCET is particularly important. In this setting, we have a new problem: to determine how the composition of components will affect the execution time. This is of course dependent on how we choose to do the composition. And this is exactly what we want to investigate in this study, a simple additive (compositional) model: take $\mathcal{T}(c)$ as an upperbound worst-case execution time of a component c, c_1 and c_2 two given components and $c_1 \otimes c_2$ their composition. Is it the case that $\mathcal{T}(c_1 \otimes c_2) = \mathcal{T}(c_1) + \mathcal{T}(c_2)$, when we consider the composition as sequential execution without interruption? What's the relation between the two sides of this equation if the equality does not hold? How is the execution of c_2 dependent on having executed c_1 first? Despite the usefulness of this model for some specific domain, it is clear that it is not valid in general due to dependencies that can be caused by the *hardware state* or the *program structure*. For example, a component can have execution paths that are not taken in a particular composition, but have to be taken into account in an analysis of the individual components, producing an overestimation

for the composite if we just add the timings. It's worth noting that this might be good enough for some soft real-time applications. In the analysis we do in this work, we take $\otimes$, the *component composition operator*, as uninterrupted sequential execution (denoted by semi-colon), and the timing $\mathcal{T}()$ based on (the results from) simulations. For example, for components c_1 and c_2, we have the equation $\mathcal{T}(c_1; c_2) = \mathcal{T}(c_1) + \mathcal{T}(c_2)$. How do hardware features and configuration affect the execution time of this composite? Some configurations might have bigger effects than others, and to understand to what extent are these effects, we have done simulations in the SimpleScalar tool chain to see how the difference in number of cycles is affected. Even though we would expect an increase in the execution time due to the extra code needed to call a service from another component, it is of good help to know which features of the architecture that cause the largest increase. In this way, we can dedicate greater effort in modelling and improving such specific features. For this purpose, we rank a subset of configurations available in the SimpleScalar tool set in section 3, where we also describe our analysis and the setup of the simulations. In section 2, we give an overview of the methodology we followed to combine the several configurations available in the tool.

2. Design of the Experiment

Statistical analysis can give more confidence in the results of experiments, and can minimize unnecessary errors, mainly when the search space is too large, allowing for a better estimation. So, rather than following an *ad hoc* choice for the hardware configuration in the simulator, we use *fractional factorial design*, as described in Yi et al. [17]. In that work, the *Plackett-Burman design* is used to analyse how changes in the architecture affect the execution time. Plackett and Burman [12] presented a methodology for the construction of very economical designs to do statistical analysis where the number of runs is a multiple of four, rather than the costly approach of running all possible combinations of the parameters. In their design, it is assumed that all interactions are negligible when compared with the important main effects. But the effects of two-parameter interactions can be quantified by using this same design with *foldover*. It still cannot identify interactions between three or more parameters.

For N parameters, the number of runs in this design is the next integer K multiple of four greater than N (for the foldover, we use $2 * K$ runs). For each run, there are specific combinations for high and low values of the parameters. For the first run, we use the high value for all the parameters. The combinations for the second run, indicating if the parameter should use its high or low value, is given by Plackett and Burman [12] for some values of K. For the next runs we do a circular shift of the combinations. In the use of foldover, we just repeat the runs, but exchanging the low and high values. As an example, if $N = 7$, then the next multiple of 4 is $K = 8$, and we have the configuration in Table 1 for each run. This kind of matrices are known as orthogonal arrays, and other methods to derive such matrices are explained in Hedayat et al. [8]. For each parameter P_i, + means the high value, and - means the low value. This same configuration is used for $N = 4$, 5 or 6, as the next next multiple of 4 for these vales is still 8 (in this case the last columns of the table are used only to do the shift). With the runs, we can easily infer the effect each parameter: the + and - in the table indicates if we should add or subtract the effect of that parameter in the run. For example, suppose that each run results in r_j units; then to find the effect of parameter P_i, we just add or subtract each value of r_j according

P_1	P_2	P_3	P_4	P_5	P_6	P_7
$-$	$-$	$-$	$-$	$-$	$-$	$-$
$+$	$+$	$+$	$-$	$+$	$-$	$-$
$+$	$+$	$-$	$+$	$-$	$-$	$+$
$+$	$-$	$+$	$-$	$-$	$+$	$+$
$-$	$+$	$-$	$-$	$+$	$+$	$+$
$+$	$-$	$-$	$+$	$+$	$+$	$-$
$-$	$-$	$+$	$+$	$+$	$-$	$+$
$-$	$+$	$+$	$+$	$-$	$+$	$-$

Table 1. Example of a configuration of runs for the Plackett and Burman design. When using foldover, we just repeat the lines and invert the signs.

to the column P_i.

3. Simulation Environment and Results

The **timing model** we want to investigate in this work is compositional and the execution time of a composite of components is given by

$$\mathcal{T}(c_1; c_2) = \mathcal{T}(c_1) + \mathcal{T}(c_2)$$

Here c_1 and c_2 are components that can be executed (or simulated) in isolation (that is the idea for components: that they should be tested and certified before used), and $c_1; c_2$ is uninterrupted sequential execution of code from the components c_1 and c_2. With this model in mind, we want to investigate how does the several features of modern computer architecture affect its precision. For this, we make simulations of the composition and of the components in isolation, and then an analysis of the difference between the two sides of this equation.

The use of **synthetic code** in the simulations makes it possible for a simple kind of components and composition: code without branches and function call without arguments, with communication through global variables. This type of code (long strips without branches, usually numerical code) is simple, and is common in programs from scientific computing, and can also be the result of optimizations like loop unrolling. The components are automatically filled with random code acting upon the global integer variables, making it quite easy to vary the size of the components. The code consists of assignment statements of basic integer arithmetic expressions to variables, with the expressions being random and containing at most ten operands (integer numbers and variables) using the basic arithmetic operators (but no division, so that we don't risk raising exceptions by dividing by zero). The simulation of this code will allow us to rank the effects of some common architecture features that are independent from the structure of the program. However, some features are of course dependant, like programs with branches and branch prediction. This kind of synthetic benchmark is of great help to give us a prior environment with results from which further investigation can be done, and of course give also initial hints about the validity of the additive timing model.

We use the SimpleScalar tool set to count the number of cycles, with the ARM architecture model (in this model, the difference in number of cycles for some applications

is less then four percent when compared with the Netwinder SA-110 hardware[1]). This tool chain is described in Burger and Austin [3] and Austin et al. [1] and has been widely used by the research community in computer architecture. The tool set accept several parameters that configure the architecture being simulated. To make use of the factorial design, we need to define the low and high value for the parameters to be used in the simulation, and here we use the same values defined by Yi et al. [17]. From the more than fourty parameters available in the simulation tool, we have chosen a total of $N = 37$ parameters that can possibly affect our timing model (for example, we left out floating point unit configuration, as we are using only integer programs). So we use $K = 40$, the next multiple of four, resulting in eighty runs (with foldover). That is, we make eighty combinations of the input parameters for the SimpleScalar, run each of them in the tool set, and record the number of cycles required to execute the program. For components $c_1; ...; c_n$, we want to know the difference between $\mathcal{T}(c_1; ...; c_n)$ and $\sum_{i=1}^{n} \mathcal{T}(c_i)$. We do this for $n =$ 2, 3, 4 and 5. Rather than using the raw value (number of cycles returned by the SimpleScalar) in the analysis, we use how big is the difference in percentage terms. For example, if $T(c_1; c_2) = x$, $\mathcal{T}(c_1) = a$ and $\mathcal{T}(c_2) = b$, we use $r = 100d/e$, where $d = |a + b - x|$ and $e = max(a + b, x)$. The result of the analysis is a table ranking each of the thirty-seven parameters, from the one that caused the biggest variation in the number of cycles, to the one that caused the smallest variation.

The plots in Figure 1 show part of the result of eighty runs for components with nine thousand lines of code (due to lack of space we show only twenty runs, at no specific order). Each of the runs have different combinations of the low and high values for the thirty-seven parameters shown in Table 2. Plot A shows the raw difference and plot B the difference in percentage terms. For this specific case, the differences are less then 1% from the total number of cycles. Using only the plots, it is not possible to know which parameter caused the big differences. Using the Plackett and Burman design, we get the data in Table 2 (last column), ranking each parameter according to its influence in causing differences in the execution time. The ranks are the average rank for components of varying sizes (1K to 9K lines of code). For a discussion of the advantages of using ranks and the choice for the low and high value for the several parameters in the table, the reader is referred to Yi et al. [17].

4. Conclusion and Future Work

In this work we investigated how some features of the microprocessor architecture affects the execution time of a program composed of components with random synthetic code. The execution time is the number of cycles returned by the SimpleScalar simulator and we used a statistical analysis (fractional factorial design) to choose which combinations of parameters to use in the simulator. With this design we ranked the parameters according to their influence on the execution time, shown in Table 2. For example, we would expect the L1 I-cache latency to have a smaller rank, i.e, have more influence on the execution time of the composition, but according to our results, it doesn't. The analysis also indicate that main memory latency and L2 cache block size have the greatest effect, and memory bandwidth has the least effect. This could have been caused by our choice for components or due to interactions between three or more parameters, as the experimental design we used cannot identify interactions of more than two parameters. We got similar results

[1]http://www.simplescalar.com/v4test.html

parameter	low value	high value	average rank
Mem. latency (first,next)	(200,4)	(50,1)	4
L2 cache block size	64	256	9.2
L1 I-cache block size	16	64	10.5
Fetch queue entries	4	32	10.5
L2 cache size	262144	8388608	11.4
Execution order	in order	out order	14.4
ROB entries	8	64	14.6
I-TLB size	32	256	14.7
L1 D-cache repl policy	l	r	15.4
L2 cache latency	20	5	15.7
I-TLB assoc	2	0	16.7
D-TLB assoc	2	0	17.3
L1 D-cache assoc	1	8	17.7
LSQ entries	2	64	17.8
L2 cache assoc.	1	8	18.5
L1 I-cache assoc.	1	2	18.5
L1 I-cache size	4096	131072	19.1
D-TLB page size	4096	4194304	20.5
D-TLB size	32	256	20.5
BTB entries	16	512	21
BTB assoc.	2	0	21.3
Branch predictor	2lev	perfect	21.4
L1 I-cache latency	4	1	21.4
int ALUs	1	4	21.5
I-TLB latency	80	t 30	21.6
L1 D-cache size	4096	131072	21.8
Spec. branch update	non-spec	decode stage	22
L1 I-cache repl policy	l	r	22.5
Int. mult-div units	1	4	22.8
Branch mispred. penalty	10	2	23
RAS entries	4	64	23.8
L1 D-cache latency	4	1	24.3
L2 cache repl policy	l	r	24.7
I-TLB page size	4096	4194304	25.2
L1 D-cache block size	16	64	25.2
Memory ports	1	4	25.5
Memory bandwidth	4	32	27.2

Table 2. Average ranks for the influence of hardware parameters in the execution time (number of cycles) of several functions with different number of lines of code. The description of each parameter can be found in [3].

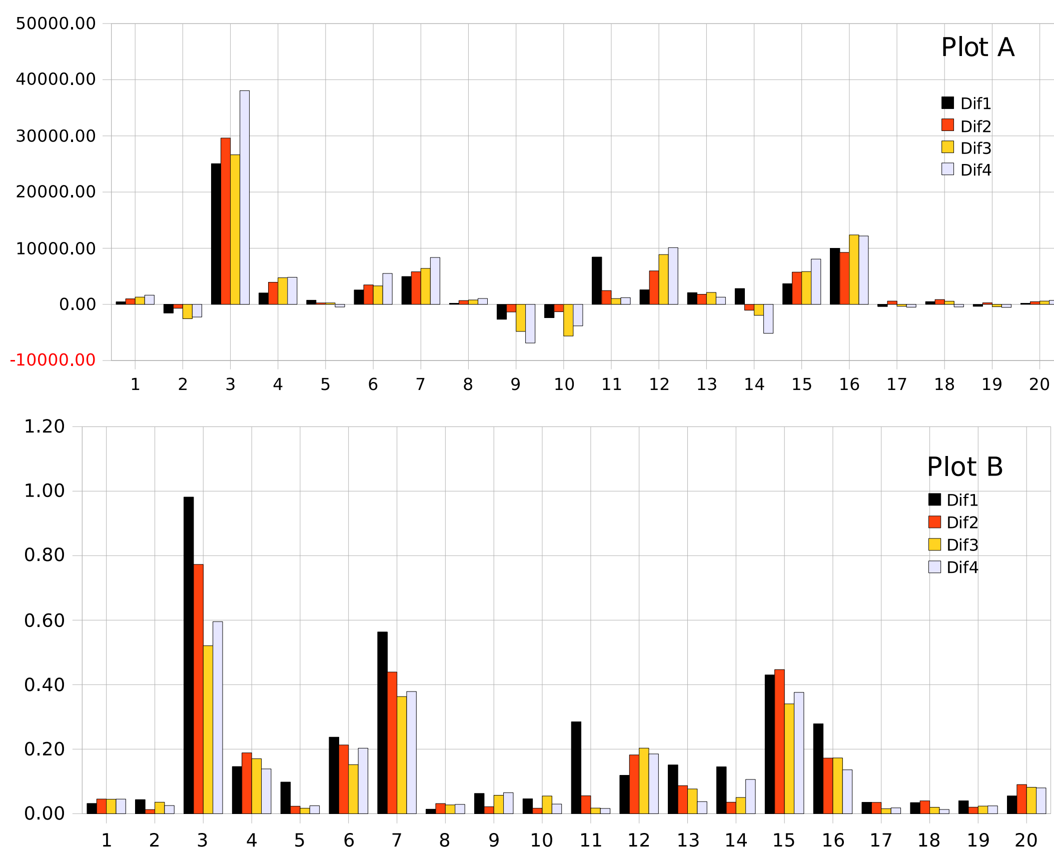

Figure 1. Example of twenty runs for components with 9K lines of code: difference in number of cycles (Plot A) and normalized (Plot B), where Dif$x$ is result of the simulation for $(x + 1)$ components.

when we did the analysis for different number of components and for components of different sizes (even though the execution time was different for the cases, the rank of the parameters were similar). The statistical analysis gives more confidence in the results, and we can focus efforts in modelling the hardware features that gives the biggest effect, and so getting a better estimation of WCET in component–based embedded systems.

The analysis also show that for the simulations with two components of size 9K lines of code each, the relative difference is between zero and one percent, meaning that for some configurations of the hardware, the additive timing model can give safe approximations for some applications. Our next step in this investigation is to adapt an embedded systems benchmark (for example, Mibench [7]) and use it in the framework of a component model (for example, Koala [14] or SaveCCM [9]).

5. Acknowledgements

This work was supported by SSF – the Swedish Foundation for Strategic Research – through the PROGRESS Research Centre for Predictable Embedded Software Systems Travel support was provided by the ARTIST2 European Network of Excellence. Special thanks to Jan Gustafsson and Andreas Ermedahl for their helpful comments.

References

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002. ISSN 0018-9162.

[2] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05)*, pages 33–40, July 2005.

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

[4] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala, Sweden, Apr. 2002. ISBN 91-554-5228-0.

[5] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17:131–181, 1999.

[6] J. Ganssle. Really real-time systems. In *Proc. of the Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, Apr. 2006.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. doi: http://dx.doi.org/10.1109/WWC.2001.15.

[8] A. S. Hedayat, N. J. A. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*. Springer-Verlag, 1999.

[9] M. Åkerholm, J. Carlson, J. Håkansson, H. Hansson, M. Nolin, T. Nolte, and P. Pettersson. The saveccm language reference manual. Technical Report MDH-MRTC-207, Mälardalen University, 2007.

[10] K.-K. Lau and S. Wang. A survey of software component models. Technical Report CSPP-30, University of Manchester, 2006.

[11] S. Lim, J. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Dec. 1998.

[12] R. L. Plackett and J. P. Burman. The design of optimum multifactorial experiments. In *Biometrika*, volume 34, pages 255–272, 1946.

[13] K. Richter. The autosar timing model – status and challenges. In *2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, 2006.

[14] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 2000.

[15] G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *Commun. ACM*, 35:89–99, Nov 1992.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem — overview of methods and survey of tools. (ISSN 1404-3041 ISRN MDH-MRTC-209/2007-1-SE), March 2007.

[17] J. J. Yi, D. J. Lilja, and D. M. Hawkins. Improving computer architecture simulation methodology by adding statistical rigor. *IEEE Trans. Comput.*, 54(11):1360–1373, 2005. ISSN 0018-9340.