WORST-CASE TIMING ESTIMATION AND ARCHITECTURE EXPLORATION IN EARLY DESIGN PHASES

Stefana Nenova and Daniel Kästner [1]

Abstract
Selecting the right computing hardware and configuration at the beginning of an industrial project is an important and highly risky task, which is usually done without much tool support, based on experience gathered from previous projects. We present TimingExplorer - a tool to assist in the exploration of alternative system configurations in early design phases. It is based on AbsInt's aiT WCET Analyzer and provides a parameterizable core that represents a typical architecture of interest. TimingExplorer requires (representative) source code and enables its user to take an informed decision which processor configurations are best suited for his/her needs. A suite of TimingExplorers will facilitate the process of determining what processors to use and it will reduce the risk of timing problems becoming obvious only late in the development cycle and leading to a redesign of large parts of the system.

1. Introduction

Choosing a suitable processor configuration (core, memory, peripherals, etc.) for an automotive project at the beginning of the development is a challenge. In the high-volume market, choosing a too powerful configuration can lead to a serious waste of money. Choosing a configuration not powerful enough leads to severe changes late in the development cycle and might delay the delivery.

Currently to a great extent this risky decision is taken based on gut feeling and previous experience. The existing timing analysis techniques require executable code as well as a detailed model of the target processor for static analysis or actual hardware for measurements. This means that they can be applied only relatively late in the design, when code and hardware (models) are already far developed. Yet timing problems becoming apparent only in late design stages may require a costly re-iteration through earlier stages. Thus, we are striving for the possibility to perform timing analysis already in early design stages.

Existing techniques that are applicable at this stage include performance estimation and simulation (e.g., virtual system prototypes, instruction set simulators, etc.). It should be noted that these approaches use manually created test cases and therefore only offer partial coverage. As a result, corner cases that might lead to timing problems can easily be missed. Since we follow a completely different approach, we will not further review existing work in these areas.

Our goal is to provide a family of tools to assist in the exploration of alternative system configurations before committing to a specific solution. They should perform an automatic analysis, be efficient and provide 100% coverage on the analyzed software so that critical corner cases are automatically

[1] AbsInt Angewandte Informatik GmbH, Science Park 1, D-66123 Saarbrücken, Germany

considered. Using these tools, developers will be able to answer questions like "what would happen if I take a core like the ABCxxxx and add a small cache and a scratchpad memory in which I allocate the C-stack or a larger cache" or "what would be the overall effect of having an additional wait cycle if I choose a less expensive flash module", ... Such tools can be of enormous help when dimensioning hardware and will reduce the risk of a project delay due to timing issues.

The paper is structured as follows. We describe how TimingExplorer is to be used in Section 2. In Section 3 we present AbsInt's aiT WCET Analyzer and compare it to TimingExplorer in Section 4. Finally we discuss a potential integration of TimingExplorer in a system-level architecture exploration analysis in Section 5.

2. TimingExplorer Usage

In the early stage of an automotive project it has to be decided what processor configuration is best suited for the application. The ideal architecture is as cheap as possible, yet powerful enough so that all tasks can meet their deadlines. There are usually several processor cores that come into question, as well as a number of possible configurations for memory and peripherals. How do we choose the one with the right performance?

Our approach requires that (representative) source code of (representative) parts of the application is available. This code can come from previous releases of a product or can be generated from a model within a rapid prototyping development environment.

The way to proceed is as follows. The available source code is compiled and linked for each of the cores in question. In the case one uses a model-based design tool with an automatic code generator, this step corresponds to launching the suitable code generator on the model. Each resulting executable is then analyzed with the TimingExplorer for the corresponding core. The user has the possibility to specify memory layout and cache properties. The result of the analysis is an estimation of the WCET for each of the analyzed tasks given the processor configuration. To see how the tasks will behave on the same core, but with different memory layout, memory or cache properties, one simply has to rerun the analysis with the appropriate description of cache and memory. Finally, the estimated execution times can be compared and the most appropriate configuration can be chosen.

Let us consider three tasks that come from a previous release of the system. Having an idea of the behavior of the old version of the system and the extensions we want to add to it in the new version, we wonder whether a MPC565 core will be sufficient for the project or it is better to use the more powerful MPC5554 core that offers more memory and cache and a larger instruction set. This situation is depicted in Figure 1. The tools we need are *TimingExplorer for MPC5xx* (*TE*$_{5xx}$) and *TimingExplorer for MPC55xx* (*TE*$_{55xx}$) for assessing the timing behavior of the tasks on the MPC565 and MPC5554 core respectively. First we compile and link the sources for MPC5xx and MPC55xx, which results in the executables E_{5xx} and E_{55xx}. Then we write the configuration files C_{5xx} and C_{55xx} to describe the memory layout, the memory timing properties and the cache parameters when applicable for the configurations in question. Next we analyze each of the tasks T_1, T_2 and T_3 for each of the two configurations. To get an estimate for the WCET of the tasks on the MPC5554 core, we run *TE*$_{55xx}$ with inputs E_{55xx} and C_{55xx} for each of the tasks. We proceed similarly to obtain an estimate of the timing for the tasks on the MPC565 core. By comparing the resulting times (X_1, X_2, X_3) and (Y_1, Y_2, Y_3), we see that the speed and resources provided by the MPC565 core will not be sufficient for our needs, so we settle for the MPC5554 core.

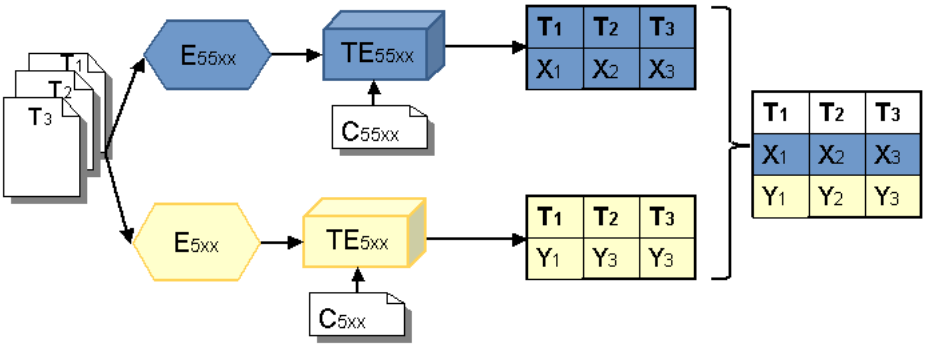

Figure 1. Usage of TimingExplorer to choose a suitable core

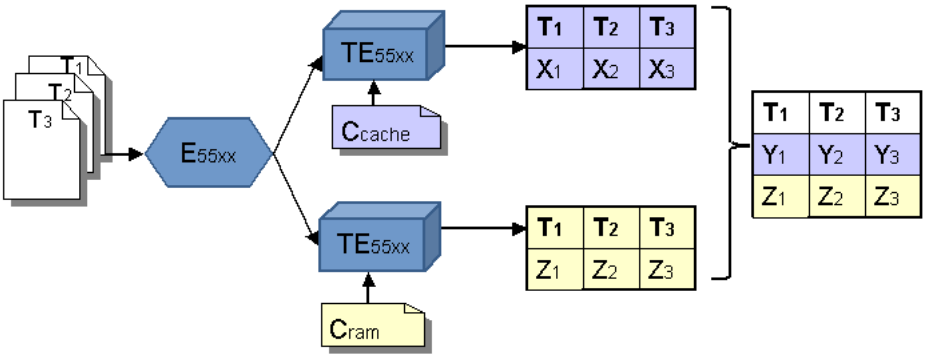

Figure 2. Usage of TimingExplorer to choose a suitable cache and memory configuration

The next step is to choose the cache and memory parameters. The core comes with 32KB cache, 64KB SRAM and 2MB Flash and gives the possibility to turn off caching and configure part of the 32KB of cache as additional RAM. We want to know what is more beneficial for our application – to have more memory or use caching. The situation is depicted in Figure 2. We write two configuration files: C_{cache}, in which we use the cache as such, and C_{ram}, in which we use it as RAM where we allocate the C stack space that usually shows a high degree of reuse. To see how the tasks run on a system with cache, we run TE_{55xx} with inputs E_{55xx} and C_{cache}. To assess the performance without the use of cache, we run TE_{55xx} with inputs E_{55xx} and C_{ram}. Based on the analysis results we can decide with more confidence which of the hardware configurations to choose.

An important question is how the choice of a compiler affects the results. Opposed to some existing simulation techniques that work solely on the source-code level and completely ignore the effect the compiler has on the program's performance, both aiT and TimingExplorer operate on the binary executable and are thus able to achieve high precision. Ideally during the architecture exploration phase the user has the compiler that would be used in the final project at his disposal (for instance because it has been used to build the previous version of the system). Using the actual compiler to obtain an executable is the best way to obtain precise estimates. However when using the production compiler is impossible for some reasons, an executable compiled with a different compiler can be used for the analysis. A preferable solution in this case is to use a compiler that is available for all configurations under consideration. In the examples discussed above we could use gcc for MPC5xx and MPC55xx to obtain E_{5xx} and E_{55xx} respectively. The goal of TimingExplorer is to assess hardware effects on the same code. The observed effects can be expected to also hold when the production compilers are not used provided that the same or at least comparable optimization settings are used.

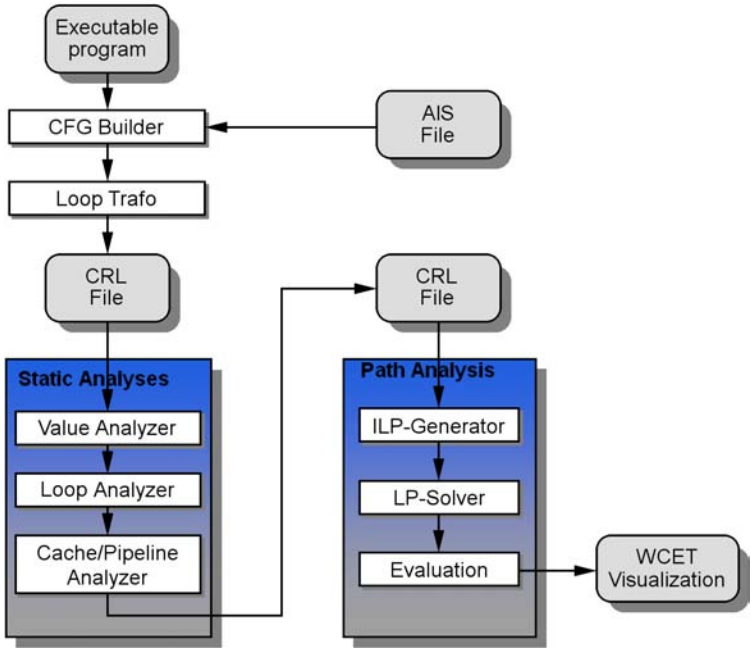

Figure 3. Phases of WCET computation

3. aiT WCET Analyzer

aiT WCET analyzer [7] automatically computes tight upper bounds of worst-case execution times of tasks in real-time systems, a task in this context being a sequentially executed piece of code without parallelism, interrupts, etc. As an input, the tool takes a binary executable together with the start address of the task to be analyzed. In addition, it may be provided with supplementary information in the form of user annotations. Through annotations the user provides information aiT needs to successfully carry out the analysis or to improve precision. Such information may include bounds on loop iterations and recursion depths that cannot be determined automatically, targets of computed calls or branches, description of the memory layout, etc. The result of the analysis is a bound of the WCET of the task as well as a graphical visualization of the call graph in which the WCET path is marked in red.

Due to the use of caches, pipelines and speculation techniques in modern microcontrollers to improve the performance, the execution time of an individual instruction can vary significantly depending on the execution history. Therefore aiT precisely models the cache and pipeline behavior and takes them into account when computing the WCET bound. In our approach [8] the analysis is performed in several consecutive phases (see Figure 3):

1. *CFG Building:* decoding and reconstruction of the control-flow graph from a binary program [20];
2. *Value Analysis:* computation of value ranges for the contents of registers and address ranges for instructions accessing memory;
3. *Loop Bound Analysis:* determination of upper bounds for the number of iterations of simple loops;
4. *Cache Analysis:* classification of memory references as cache misses or hits [6];

5. *Pipeline Analysis:* prediction of the behavior of the program on the processor pipeline [14];
6. *Path Analysis:* determination of a worst-case execution path of the program [21].

aiT uses abstract interpretation [2] to approximate the WCET of basic blocks (steps 2, 4 and 5) and integer linear programming to compute the worst case execution path (step 6).

Furthermore within the INTEREST project [13] aiT has been integrated with two model-based design tools – ASCET [5, 10] and SCADE [4, 9], which are widely used in the automotive and avionic domain respectively. Both tools provide automatic code generators and allow their users to start an aiT WCET analysis from within the graphical user interface, providing them with direct feedback on the impact of their design choices on the performance.

4. From aiT to TimingExplorer

4.1. Overview

aiT is targeted towards validation of real-time systems, so it aims at high precision and closely models the underlying hardware. However in early design phases one might want to investigate the behavior of code on an architecture that closely resembles a certain existing microcontroller for which no high-precision model is (yet) available. Therefore a flexible, yet simple way to specify the memory layout and behavior is necessary. Such a specification should make it possible to run analysis for microcontrollers that are still in development and for which no hardware models exist yet. The way cache and memory properties are specified in TimingExplorer is described in Section 4.2.

Furthermore aiT is sound in the sense that it computes a safe overapproximation of the actual WCET. Depending on the processor this can lead to high analysis times (typically up to several minutes for a task). Such analysis times are acceptable in the verification phase, but are undesirable for configuration exploration. Moreover as opposed to verification for which soundness is of utmost importance, for dimensioning hardware an overapproximation is not always necessary. As long as the tool reflects the task's timing behavior on the architecture and allows comparison of the timing behavior on different configurations, slight underapproximations are acceptable.

Therefore some precision is traded against ease of use, speed and reduced resource needs e.g., through performance optimizations concerning the speed and memory consumption of the pipeline analysis (Section 4.3). Improved usability will be achieved by incorporating automatic source-level analyses that will reduce the need for manual annotation. The types of source-level analyses we are currently working on are described in Section 4.4.

4.2. Cache and Memory Specification

To enable the user to experiment with different configurations in the architecture exploration phase, we have made the cache and memory mapping in TimingExplorer completely parameterizable through annotations. It is possible to set the cache size, line size, replacement policy and associativity independently for the instruction and data cache. Furthermore one can choose how unknown cache accesses are to be treated – as cache hits or cache misses. For example the following specification describes two L1 caches, a 16 KB instruction and a 32 KB data cache that both use an LRU replacement policy:

```
cache instruction
  set-count = 128, assoc = 4, line-size = 32,
  policy = LRU, may = empty
and data
  set-count = 256, assoc = 4, line-size = 32,
  policy = LRU, may = empty;
```

With respect to the memory map, one can specify memory areas and their properties. A memory area is specified as an address range. Useful properties include the time it takes to read and write data to the area, whether write accesses are allowed or the area is read-only, if accesses are cached, etc.

4.3. Pipeline Analysis

To speed up the analysis and reduce memory needs TimingExplorer uses parametric pipeline analysis and computes local rather than global worst-case time as done by aiT. The difference between the latter two kinds of computations is what the tool does when the abstract cache and pipeline state is about to fork into two or more successor states because of imprecise information. This happens for instance when a memory access cannot be classified as cache hit or cache miss. In the case of global worst-case, the tool creates all successor states and follows their further evolution. In contrast, local worst-case means that it immediately decides which successor is likely to be the worst, and only follows the evolution of this single state.

For example, consider that we have two consecutive unclassified memory accesses a_1 and a_2 and let h_i denote the case when a_i is a cache hit and m_i when it is a cache miss. aiT uses global worst-case computation, so after the first unclassified access it creates the superstate $\{(h_1), (m_1)\}$ and after the second – the superstate $\{(h_1, h_2), (h_1, m_2), (m_1, h_2), (m_1, m_2)\}$. From that point on it performs every computation for each of the four concrete states. At the end it compares the resulting times and chooses the highest one. Due to timing anomalies it is not necessary that the worst case is the one resulting from the state (m_1, m_2).

On the other hand TimingExplorer uses local worst-case computation. Unless anything else is specified, this means that a memory access takes longer to execute if it is a cache miss. Therefore after the first unknown access TimingExplorer will simply assume that it is a cache miss and create the state (m_1) and after the second the state (m_1, m_2). It proceeds the execution with a single state (and not with a superstate like aiT).

It should be noted that because of the local worst-case computation the result is not necessarily an overapproximation of the WCET. It is sound in the case of fully timing compositional architectures i.e. architectures without timing anomalies [22], but can be an underestimate otherwise. However the use of local worst case may drastically improve the performance in the cases when there are frequent pipeline state splits.

4.4. Source-code Analyses

4.4.1. Overview

To successfully analyze a task, aiT and TimingExplorer need information about the number of loop iterations, recursion bounds, targets of computed calls and branches. Partially such information can

be automatically discovered when analyzing the binary, e.g., the loop bound analysis mentioned in Section 3 can discover bounds for the number of iterations of simple loops. However information that cannot be discovered automatically has to be supplied to the tool in the form of annotations. To reduce the need of manual annotation, source level analysis will be incorporated that will try to determine auxiliary information that TimingExplorer can extract from the executable.

It should be noted that due to compiler optimizations, the information extracted from the source code does not necessarily correspond to the behavior of the executable. Consider the loop

```
for (int i=0; i<100; i++)
  ...
```

The compiler could discover that the loop will be executed at least once and move the exit condition from the header to the bottom, turning it into a do/while loop. Doing so it reduces the number of jumps by two, which can improve the performance, since jumps often cause a pipeline stall. Furthermore the test condition will be evaluated 100 times (and not 101 times as we will discover by source-level analysis). The resulting overapproximation will result in some precision loss. Whereas in a verification phase the analysis precision is of utmost importance and any loss of precision is undesirable, in an exploration configuration stage the overapproximation is often acceptable because of the time and effort gain from reducing the manual annotation process. Furthermore, the results of the source-level analyses are independent of the compiler and the target architecture and therefore the resulting precision losts will be the same for each of the analyzed configurations. Thus despite the possible overapproximation introduced by incorporating source-level analyses, the comparison of different configurations will still be precise.

Currently, we are working on source level analyses for extracting loop bounds [12] and targets of function calls when function pointers are used [18]. Both of these analyses are implemented with help of the SATIrE framework [17], the LLNL-ROSE compiler [15] and PAG [16].

4.4.2. Function-pointer Analysis

The use of function pointers makes the static analysis of a program hard, because in the general case it cannot be determined which functions will be called at runtime. This leads to an imprecise control flow graph and therefore imprecise results. Unfortunately function pointers are quite common in automotive software and manual annotation should be avoided, because it can be cumbersome and error-prone. Therefore we plan to integrate an automatic analysis for function pointer resolution.

For each function pointer variable in the program the analysis computes a points-to set containing all memory locations whose address might be stored in the variable. The analysis information is stored in a recursive fashion thus making it easy to handle function pointers that are stored within arbitrarily complex objects, like structures with arrays of function pointers. As an output the analysis can create annotations for TimingExplorer, which will relieve the user from determining targets of function pointer calls manually.

During our experiments we were able to successfully analyze several medium-sized C programs. An overview of our test suite is given in Table 1. For the programs in this suite, the analysis detected all function pointers and a manual inspection of the results gave the impression, that they were close to an optimal solution. Our first attempts to analyze real automotive software were not successful,

Program	Version	Lines of code	Indirect calls
diction	0.7	2 037	3
grep	2.0	12 417	3
gzip	1.2.4	8 163	4
sim6890	0.1	3 290	97

Table 1. Test suite for the source-level analysis of function pointers

primarily because of problems in LLNL-ROSE and SATIrE. In some cases this was caused by the use of special language keywords, since the projects had been developed using compilers for target architectures with small word size and used special language constructs. Unfortunately SATIrE also had problems with certain C-constructs (e.g., valid source made it crash).

4.4.3. Loop-bound Analysis

In the case of loop bound analysis we express the upper loop bound as a parametric formula depending on *parameters* – variables and expressions that are constant within the loop. The symbolic formula is constructed once for each loop in an analysis run that precedes the rest of the analyses. Afterwards as soon as the value analysis obtains bounds on the values of all the parameters in a certain formula, a concrete bound on the loop iteration count for that case can be computed with low computational effort.

In an initial evaluation phase we used the Mälardalen WCET benchmark suite [23] to compare our analysis to oRange [1] and the loop bound analysis component in SWEET [3]. The tests showed that:

- The analysis detects and assigns a loop bound to all loops established with C loop constructs except for the cases of loops inside recursive functions or when LLNL-ROSE and SATIrE meet unsupported constructs.
- The precision of the computed loop bounds (assuming that precise values of parameters are provided by an external value analysis) is higher than the one of SWEET and in most cases higher or equal to the precision of oRange.
- The analysis can cope with automatically generated code.
- The analysis is especially effective for typical counter-based loops, which are common in generated code.

Therefore we believe that incorporating source-level loop-bound analysis in TimingExplorer will drastically reduce the number of loops that have to be annotated manually. In the case, when even after the automatic analysis some loop bounds are unknown and the user does not want to spend time on specifying their bounds manually, we have foreseen the possibility to give a global default bound to all loops of unknown iteration count.

5. Integration in a System-level Analysis

After incorporating the source-code level analyses discussed in the previous section in TimingExplorer, we will have a tool capable of predicting the WCET of a task on a selected target architecture

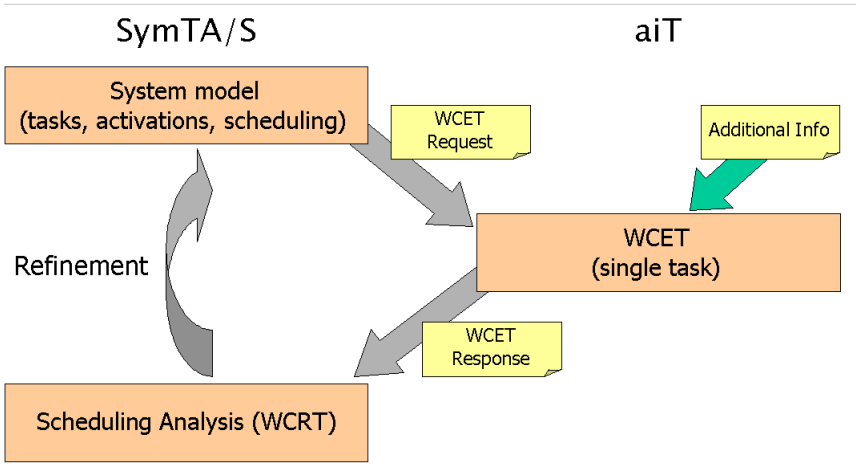

Figure 4. Flow of requests and responses between aiT and SymTA/S

almost automatically. Yet as presented so far, the tool will work only for single tasks without interrupts or exceptions. On the other hand, schedulability analyzers can determine the overall worst-case response time of a system provided the corresponding results for single tasks are known.

Within the European FP6 STREP Project INTEREST [13], a generic information exchange format called XML Timing Cookies (XTC) has been developed. With its help, users of a scheduling analysis tool can cause aiT to perform timing analyses at the code level, whose results are mapped back to the scheduling tool in a fully automatic way. In particular a coupling of aiT with SymTA/S has been realized (Figure 4).

SymTA/S [11, 19] of Symtavision GmbH is a modular tool-suite for scheduling analysis and optimization for electronic controllers, buses/networks and complete embedded real-time systems. It computes the worst-case response times of tasks and the worst-case end-to-end communication delays taking into account the WCETs of the tasks, scheduling, bus arbitration, possible interrupts and their priorities. Two SymTA/S modules that are of special interest for architecture exploration are the *Sensitivity Analysis* and *Design-Space Exploration*. The Sensitivity Analysis module allows its user to determine the robustness of a system and its extensibility for future functions. With the help of the Design-Space Exploration plug-in one can evaluate alternative system configurations and perform system optimization. One specifies which system parameters can be varied and defines optimization objectives, and the tool presents him the most interesting alternatives and the corresponding trade-offs.

We plan to use XTC and integrate TimingExplorer into the SymTA/S system-level architecture exploration analysis. After such integration an overall timing analysis can be performed that will allow users to assess how much room there is for estimation errors and how much flexibility remains for later changes. The combination of code-level and system-level architecture exploration will lead to informed decisions with respect to which architectures are appropriate for an application.

6. Conclusion

We have presented TimingExplorer – a source-level worst-case timing estimation tool to assist in the exploration of alternative system configurations in early design phases. It is an extension of

AbsInt's aiT WCET Analyzer, provides a flexible way to specify an architecture of interest and yields precise estimates of the time it will take a task to run on the modelled architecture. Furthermore TimingExplorer takes all program executions on all inputs into account so that critical corner cases are automatically considered.

We gave an example how a suite of TimingExplorers is to be used to decide what architecture to use for a project and explained the way TimingExplorer differs from aiT. We described two types of source-code analyses that we want to incorporate in TimingExplorer to minimize the need for manual annotation. We also suggested a way to integrate the tool in a system-level exploration framework so that the overall timing performance of the system can be estimated.

We believe that a suite of TimingExplorers will considerably facilitate the process of dimensioning hardware at the beginning of a project. It will allow system architects to assess performance considerations, such as processor capacity, memory layout, cache sizing etc. Thus when choosing what configuration to use they will no longer rely solely on experience and intuition as they do now, but will have an estimate of how the software will behave on the selected architecture at hand.

7. Acknowledgements

This work has partially been supported by the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Program.

References

[1] CASSÉ, H., DE MICHIEL, M., BONENFANT, A., AND SAINRAT, P. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *Proceedings of the 14th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008)* (Kaohsiung, Taiwan, 2008).

[2] COUSOT, P., AND COUSOT, R. Abstract Interpreatation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977)* (Los Angeles, California, 1977).

[3] ERMEDAHL, A., SANDBERG, C., GUSTAFSSON, J., BYGDE, S., AND LISPER, B. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis (WCET 2007)* (Pisa, Italy, 2007).

[4] Esterel Technologies. SCADE Suite. `http://esterel-technologies.com/products/scade-suite`.

[5] ETAS Group. ASCET Software Products. `http://www.etas.com/en/products/ascet_software_products.php`.

[6] FERDINAND, C. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[7] FERDINAND, C., AND HECKMANN, R. aiT: Worst-Case Execution Time Prediction by Static Programm Analysis. In *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions*. Toulouse, France, 2004, pp. 377–384.

[8] FERDINAND, C., HECKMANN, R., LANGENBACH, M., MARTIN, F., SCHMIDT, M., THEILING, H., THESING, S., AND WILHELM, R. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the First Workshop on Embedded Software (EMSOFT 2001)* (Tahoe City, CA, USA, 2001), vol. 2211 of *Lecture Notes in Computer Science*, Springer-Verlag.

[9] FERDINAND, C., HECKMANN, R., LE SERGENT, T., LOPES, D., MARTIN, B., FORNARI, X., AND MARTIN, F. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In *Proceedings of the 4th European Congress Embedded Real Time Software (ERTS 2008)* (Toulouse, France, 2008).

[10] FERDINAND, C., HECKMANN, R., WOLFF, H.-J., RENZ, C., GUPTA, M., PARSHIN, O., AND WILHELM, R. Towards integrating model-driven development of hard real-time systems with static program analyzers. In *SAE 2007* (Detroit, USA).

[11] HENIA, R., HAMANN, A., JERSAK, M., RACU, R., RICHTER, K., AND ERNST, R. System level performance analysis – the SymTA/S approach. In *IEEE Proceedings Computers and Digital Techniques* (2005), vol. 152.

[12] HONCHAROVA, O. Static Detection of Parametric Loop Bounds on C Code. Master's thesis, Saarland University, 2009.

[13] INTEREST project. `http://www.interest-strep.eu`.

[14] LANGENBACH, M., THESING, S., AND HECKMANN, R. Pipeline modeling for timing analysis. In *Proceedings of the 9th International Static Analysis Symposium (SAS 2002)* (Madrid, Spain, 2002), vol. 2477 of *Lecture Notes in Computer Science*, Springer-Verlag.

[15] LLNL-ROSE. `http://rosecompiler.de`.

[16] MARTIN, F. *Generation of Program Analyzers*. PhD thesis, Saarland University, 1999.

[17] SCHORDAN, M. Combining tools and languages for static analysis and optimization of high-level abstractions. Tech. rep., TU Vienna, Austria, 2007.

[18] STATTELMANN, S. Function Pointer Analysis for C Programs. Bachelor's thesis, Saarland University, 2008.

[19] Symtavision. SymTA/S. `http://www.symtavision.com/symtas.html`.

[20] THEILING, H. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing and Applications Symposium (RTCSA 2000)* (Cheju Island, South Korea, 2000), IEEE Computer Society.

[21] THEILING, H., AND FERDINAND, C. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)* (Madrid, Spain, 1998).

[22] THIELE, L., AND WILHELM, R. Design for timing predictability. *Real-Time Systems 28*, 2-3 (2004), 157–177.

[23] WCET benchmarks project. `http://www.mrtc.mdh.se/projects/wcet`.