A Retargetable Software Timing Analyzer Using Architecture Description Language

Xianfeng Li[†] Abhik Roychoudhury[‡] Tulika Mitra[‡] Prabhat Mishra[*] Xu Cheng[†]

[†]Dept. of Computer Sc. & Tech.
Peking University
Beijing, P.R. China, 100871
lixianfeng@mprc.pku.edu.cn
chengxu@mprc.pku.edu.cn

[‡]School of Computing
National University of Singapore
Republic of Singapore, 117543
abhik@comp.nus.edu.sg
tulika@comp.nus.edu.sg

[*]Computer and Information Sc. & Eng.
University of Florida
Gainesville, FL 32611 USA
prabhat@cise.ufl.edu

Worst Case Execution Time (WCET) is an essential input for performance and schedulability analysis of real-time systems. Static WCET analysis requires program path analysis and microarchitecture modeling. Despite almost two decades of research, WCET analysis has not enjoyed wide acceptance in industry. This is in part due to the difficulty in microarchitecture modeling of modern processors. Given the large number of embedded processors available in the market, retargetability of the WCET analysis framework is a serious issue. In this paper, we address it using Architecture Description Language (ADL). Starting with the ADL of a target processor, the proposed framework automatically generates graph-based execution models to capture timing effects of instructions in the pipeline. This pipeline model coupled with parameterized models of cache and branch prediction lead to a WCET framework that is safe, accurate and retargetable.

I. Introduction

Schedulability analysis of real-time embedded systems requires the worst-case execution time (WCET) of each task. WCET of a task is the maximum possible execution time of a task *for all possible inputs* on a particular processor platform. Significant research effort has been invested to develop static timing analysis techniques for estimating tight upper bounds on the WCET. Clearly, such techniques must take into account program path information and the timing effects of the underlying architecture (e.g., pipeline, cache, branch prediction). Modeling modern architectural features for WCET analysis is not an easy task. A decade of research has resulted in WCET modeling techniques for various complex architectures. Still, developing a timing analyzer for a new platform is an extremely time-consuming effort. Unlike the desktop domain, which is dominated by the x86 architecture, the embedded domain has a large number of possible architectural platforms to choose. This, coupled with the strict time-to-market pressure for embedded systems, has prevented wide acceptability of software timing analyzers in the industry. The current practice is to use simulation-based techniques, which cannot guarantee the safety of WCET estimates. This is not an acceptable solution for safety-critical embedded systems such as automotive, avionics, and medical applications.

Clearly, retargetability is one of the most important issues for increasing the acceptability of software timing analyzers in industry. To achieve this, we need a formal description of the target processor architecture. Recently, various Architecture Description Languages (ADLs) have emerged. ADLs describe the instruction-set architecture (ISA) as well as the microarchitecture of a given platform [18, 4]. Researchers have studied the retargetability issues in the context of ADL-driven compilation [15, 18], simulation [14], hardware synthesis [16] as well as functional validation [12]. In this paper, we explore the possibility of ADL-driven retargetable timing analysis for embedded real-time software. *To our knowledge, there are no published results on retargetable static WCET analysis of processor pipeline using ADLs.*

Certain architectural features are easy to retarget as they are highly parameterizable. Examples include instruction cache size, line size, associativity, etc. Most WCET tools can handle such parameters. We are interested in exploring the possibility of retargeting features that are not easily parameterizable, in particular pipeline, by exploiting ADL.

Our timing analyzer is based on a graph-based model, called the *execution graph* [10], to capture the pipeline behaviors. The nodes of the graph denote the pipeline stages of the instruction in a code fragment. The edges denote execution dependences between instructions. Resource contentions and other relations among instructions that affect their execution are also modeled. The execution graph forms a powerful intermediate representation that the WCET analyzers can work with, and it is also an excellent representation for easy retargetability. Since the timing analysis algorithm works on the execution graph, architectural changes are transparent to it. Thus the challenge in developing a retargetable timing analysis framework is to *automatically* produce such execution graphs from the ADL specification of a processor. In this work, we present an approach to fulfill this task. As a concrete demonstration, we develop a retargetable WCET analyzer based on the EXPRESSION ADL [4] that captures the instruction set architecture (ISA) and the microarchitecture of a processor.

II. Related Work

Research on WCET analysis was initiated more than a decade ago. Early activities can be traced back to [13, 17].

These works analyzed the program source code but did not consider hardware features such as cache or pipeline. Subsequently, many researchers have investigated the issue of modeling timing effects of micro-architectural features such as instruction/data cache [11, 19], in-order/out-of-order pipelines [3, 9, 10] and branch prediction [2, 8]. Recently WCET analysis has been employed on real-life modern processors [7, 6].

Retargetable WCET analysis tools have been studied in [1, 5, 20]. The work of [1] is closer to ours. To avoid handcrafted code for microarchitecture modeling, the authors use MESCAL and its architecture description language called MADL. MADL contains both ISA information and microarchitecture description, and MESCAL can construct compilers and simulators automatically from an architecture described in MADL. They obtain the WCET for a code fragment by simulation. As mentioned earlier, simulation does not guarantee a safe bound. Indeed, MESCAL is primarily targeted to VLIW processors, and with certain restrictions on pipeline complexities, it is possible to obtain the WCET by simulation. In contrast, our microarchitecture modeling is based on static analysis that suffers no such restrictions.

The work of [5] uses a technique called micro-analysis. It transforms machine instructions into a sequence of primitive operations, on which timing analysis is conducted. The timing analysis is guided by a set of pattern-driven rules. Although the primitive operation language is architecture-independent, targeting the analysis to an architecture needs a set of timing rules for that architecture to be constructed, and this has to be done manually. Therefore, it requires a deep understanding of the timing model of the processor, and the retargeting effort is non-trivial. Furthermore, timing rules are not powerful enough to capture complicated instruction interactions in modern processor pipelines.

The work of [20] takes in low-level Hardware Description Language (HDL) specifications of pipelines and defines pipeline analysis as computations on FSMs. Even with the help of binary decision diagram (BDD), it still suffers from the problem of state space explosion. For example, the author admitted that MPC755, an embedded processor with some features of modern superscalar procesors, poses a challenge because of the huge state space of the pipeline model. Our pipeline model is based on an execution graph representation which captures potential dependences and resource contentions among interacting instructions in the pipeline. This is a much more compact pipeline model. In addition, this representation closely resembles datapaths described in ADLs. Thus constructing execution graphs automatically from ADLs is feasible.

III. Overview

In this section, we present an overview of the retargetable analyzer. Conceptually, WCET analysis performs two tasks: program path analysis and microarchitecture modeling. Program path analysis is responsible for finding the worst case execution path, and microarchitecture modeling determines instruction timing. Once the two sets of information are available, the WCET can be calculated. In this work, we adopt an Integer Linear Programming (ILP) based WCET framework [8, 11], which formulates WCET calculation as an ILP problem as follows.

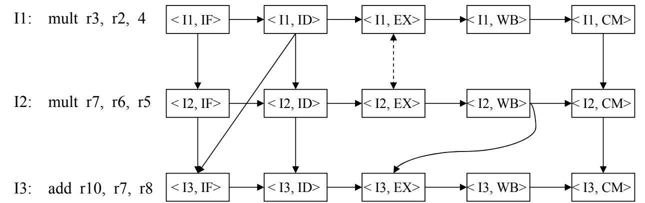

Fig. 1. Example execution graph

$$\text{maximize} \sum_{B \in \mathcal{B}} N_B * c_B$$

where N_B is a variable denoting the execution count of basic block B and c_B is a constant denoting the WCET estimate of B. Linear constraints on N_B are developed from the control flow graph,

$$\sum_{B' \to B} E_{B' \to B} = N_B = \sum_{B \to B''} E_{B \to B''}$$

where $E_{B' \to B}$ denotes the number of times control flows through the edge $B' \to B$.

It can be seen that only c_B is affected by the underlying hardware. The following section describes how c_B is estimated *automatically* from the pipeline model described by an ADL.

A. Basic Block Timing Analysis

In [10], the execution of a basic block B on a pipeline is modeled with an *execution graph*. The nodes in the execution graph correspond to pipeline stages of the instructions in B. For example, $\langle I_1,\ IF \rangle$ is I_1 (the first instruction of B) in its instruction fetch (IF) stage. There are three types of relations among the graph nodes that affect their execution: dependence relation, contention relation, and parallel relation.

Dependence relation Given two nodes u and v, if v can start execution only after the completion of u, then a dependence relation exists between them. This dependence is indicated by a solid directed edge from u to v in the execution graph. A dependence can be one of the four cases.

- Dependence among stages of the same instruction. For example, $\langle I,\ IF \rangle \to \langle I,\ ID \rangle$ means instruction decode must follow instruction fetch.
- Dependence due to in-order execution in some stages. For example, $\langle I_i,\ IF \rangle \to \langle I_{i+2},\ IF \rangle$ is drawn for a processor with 2-wide fetch.
- Dependence due to limited buffer capacity. For example, $\langle I_i,\ ID \rangle \to \langle I_{i+4},\ IF \rangle$ is drawn for a processor with 4-entry fetch buffer because the fetch buffer has no entry to accommodate I_{i+4} before the decode of I_i.
- Data dependence among instructions. For example, if the result of I_i is used by I_j, then a dependence edge $\langle I_i,\ WB \rangle \to \langle I_j,\ EX \rangle$ is drawn, assuming the result is produced in the write-back stage of I_i, and it is needed in the execution stage of I_j.

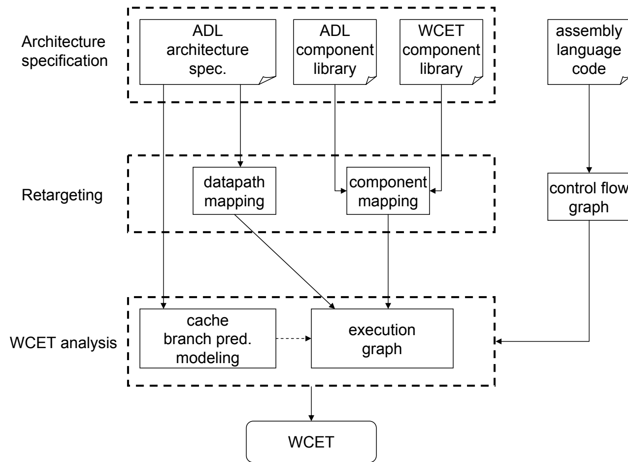

Fig. 2. Retargetable WCET analysis framework

Contention relation In out-of-order processors, instructions may delay each other because of resource contention. For example, for two independent instructions I_i and I_j that both need the ALU in the execution stage, if they can coexist in the pipeline, then a contention relation between $\langle I_i,\ EX\rangle$ and $\langle I_j,\ EX\rangle$ is recorded.

Parallelism relation In superscalar processors, an instruction's execution in a pipeline stage may be delayed due to limited superscalarity, e,g., even if the dependences and contentions of $\langle I,\ EX\rangle$ are cleared, it may still need to wait because of limited issue width. Parallelism relation is defined to consider this problem. If two independent instructions can execute in parallel in a stage, then a parallelism relation is recorded.

With the constructed execution graph, the worst case execution time of B can be estimated by considering the aforementioned relations. The details of the algorithm is presented in [10]. Since the estimation algorithm works on the execution graph, it is architecture-independent and can be targeted to different processor models without changes. However, the construction of the execution graph is based on an in-depth understanding of the pipeline model, which is manual and error-prone. In this paper, we develop a framework to automate this process.

B. Retargetable Framework

The retargetable WCET analysis framework is presented in Fig.2. It takes as input the architecture specification described in an ADL and the assembly code of the program, and automatically generates WCET estimate of the program by considering the timing effects of the processor architecture expressed in ADL.

In this work, we use EXPRESSION [4]. The ADL specification consists of two parts: instruction set architecture and microarchitecture. The microarchitecture description again has two parts: structural and behavioral descriptions. The structural part is in the main ADL description, while the behavioral description appears in the so-called *ADL component library* (see Fig.2). The component library is written in a traditional language like C++. For example, in an example provided by the EXPRESSION group, the component library contains FetchUnit, ExecuteUnit, LoadStoreUnit, etc.

Execution graph construction is performed in two steps. The first step is to construct the graph nodes for each instruction. This corresponds to *datapath mapping* and *component mapping* in Fig.2. Datapath mapping converts processing elements on pipeline paths to execution graph nodes; component mapping annotates each graph node with properties needed in the next step. Given the execution graphs nodes, the second step is to construct their relations, including dependences, contentions, and parallelism relations. This is achieved by examining the properties (input/output storage elements) of the graph nodes. For example, if node v's output storage is node u's input storage, and no other node between them accesses this storage, then a dependence edge representing the producer-consumer relation is constructed.

IV. From ADL to Execution Graph

In this section, we elaborate on the process of constructing the execution graph of a basic block from an ADL specification automatically. We begin with a description of the abstract WCET component library.

A. WCET Component Library

To facilitate a generalization of the execution graph and its automatic construction, the first thing is to design a set of abstract component models which are independent of a specific pipeline and are suitable for WCET analysis.

According to our observation, the ADL specification, although at a much higher level of abstraction, is similar to an RTL description in that a pipeline path consists of a series of processing elements interfaced by storage elements. Moreover, the behaviors of the processing elements are often dictated by their input/output storage elements. Therefore, instead of abstracting on the processing elements, we focus on the storage elements.

Array model An array model implies indexed accesses. Typical examples are register files and caches. The primary timing property for the array model is its access latency. As an example, the impact of an instruction cache with a latency of T cycles on the execution graph is just a delay of T associated with the dependence edge

$$\langle I,\ IF\rangle \rightarrow \langle I,\ stage\rangle$$

where I is the instruction identifier and $stage$ is the stage following the fetch stage in the pipeline (typically decode).

FIFO model FIFO model refers to storage elements with first-in first-out policy. FIFO models are typically associated with in-order pipeline stages. A latch can be viewed as a special FIFO buffer with a single entry. An example FIFO model is the instruction fetch buffer. A storage element with FIFO

model has very different timing effects compared to the array model. It can be characterized by three parameters: size, parallelism (number of instructions that can enter/leave the FIFO in parallel), and latency. There are three possible dependences between FIFO producers/consumers, which we illustrate with the help of instruction fetch buffer.

- **Producer-consumer:** $\langle I,\, IF\rangle \rightarrow \langle I,\, ID\rangle$
- **Parallelism limit:** $\langle I_i,\, IF\rangle \rightarrow \langle I_{i+p},\, IF\rangle$, and $\langle I,\, ID\rangle \rightarrow \langle I_{i+p},\, ID\rangle$ where the instruction buffer's parallelism is p. It means I_{i+p} cannot be fetched or decoded together with I_i.
- **Size limit:** $\langle I_{i-size},\, ID\rangle \rightarrow \langle I_i,\, IF\rangle$. This captures the fact that the fetch will stall if the buffer is full.

Pool model A pool model refers to a buffer with out-of-order (i.e, non-FIFO) dequeue process. A typical example is the issue queue in an out-of-order processor, where instructions are placed in-order by the decode unit, but may be issued from it out-of-order. Capturing the timing effects of pool model for WCET analysis is in general difficult due to its highly unpredictable behavior. Nevertheless, a subset of pool models can be modeled with limited retargetability. In [10], a buffer which serves as both an issue queue and a reorder buffer is modeled. This kind of pool model is included in the WCET component library. For a pool model that cannot be mapped to the WCET component library, we resort to handcrafted WCET analysis code.

B. Execution Graph Construction

We discuss the execution graph construction in three steps: preparation, graph node construction, and graph relation construction.

Preparation Before constructing any execution graph, a set of data structures are built.

The first data structure is a table that maps ADL component classes to WCET component models. Typically, an ADL description contains only a small number of component classes, such as FetchUnit, DecodeUnit, ExecuteUnit, BranchUnit, etc. A component class may have multiple instances, called components. For example, a processor may have multiple ALUs, which are instances of ExecuteUnit. The functions and behaviors of component classes are not described in ADL; instead, they are described in a library written in traditional languages like C++. Because of this, we need to understand these component classes by reading C++ code, then decide their corresponding WCET component classes. This is the only manual work.

The second data structure is the set of pipeline datapaths. It is generated by parsing the PIPELINE section of the ADL description, which specifies the pipeline datapaths in a hierarchical form. The case study in Section V will give such an example.

The third data structure is a datapath map table. For each opcode in the ISA, the analyzer generates an entry in the table that maps the opcode to a set of datapaths. To generate this table, the analyzer reads the OPCODES field of each datapath component in the ARCHITECTURE section of the ADL description. The OPCODES field contains the groups of opcodes that may use this component. A datapath is mapped to an opcode if all components on the datapath allow this opcode to proceed through.

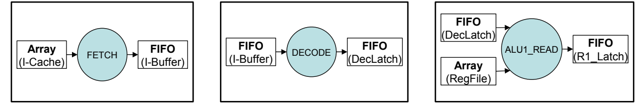

Fig. 3. Example execution graph nodes

Graph node construction Given a sequence of instructions, the analyzer generates the corresponding graph nodes with the help of aforementioned tables.

When processing an instruction, its opcode is used to look up the datapath map table to get the mapped datapaths. Then, corresponding paths consisting of execution graph nodes (without edges connecting them) are generated. During this process, one important work is component mapping that decides the WCET component model for each graph node on a path. This is done by looking up the component map table using the corresponding ADL component class as index. After component mapping, the graph nodes are annotated with properties that are needed for constructing node relations.

Fig.3 shows three example graph nodes, which correspond to the FetchUnit, DecodeUnit, and OpReadUnit respectively. The circle in a graph node represents the processing element; the left/right rectangles with directly edges to/from a circle are input/output storage elements. In the analyzer, two fields are maintained for each storage element: a WCET component model (array, FIFO, etc), and a pointer to the storage (I-Cache, I-Buffer, etc). The storages are maintained by the analyzer as another set of data structures in addition the graph nodes. They will be needed in graph relation construction.

Graph relation construction To construct graph relations, the graph nodes are processed in program order, i.e., from the nodes of the first instruction to those of the last instruction in the sequence. And the nodes of the same instruction are processed in the order of pipeline stages. This processing order is to ensure correct producer-consumer relation as well as other relations.

To find out a graph node's relations with others, the analyzer needs to know (1) the WCET component models (array, FIFO, etc) of the input/output storage elements, and (2) the nodes that read or write its storage elements. The first set of information has been obtained during graph node construction. The second set of information can be collected by the in-order traversing of graph nodes without any human intervention. Take the graph nodes in Fig.3 for example. When processing the FETCH node, the analyzer updates I-Buffer by recording the FETCH node as its latest producer. Next, when processing the DECODE node of the same instruction, the analyzer finds out that its input storage is a FIFO model, and according to the rule

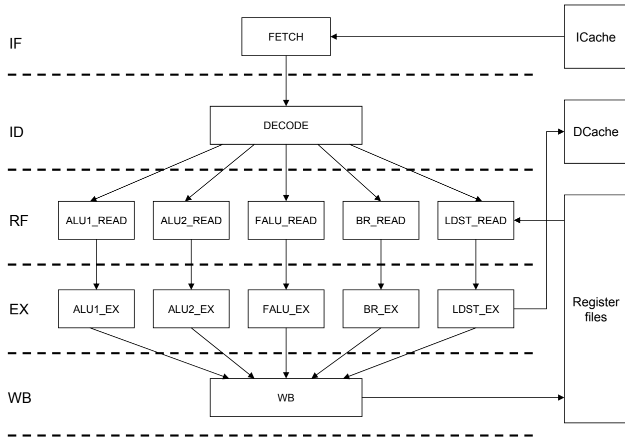

Fig. 4. Architecture block diagram of acesMIPS

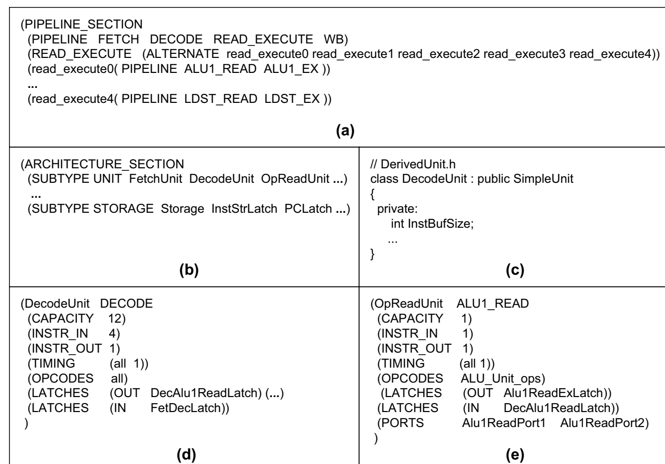

Fig. 5. Part of the acesMIPS ADL description

of a FIFO model, a producer-consumer relation should be built between the producer of the first available data in the FIFO and the DECODE node. Since the FIFO points to I-Buffer, whose latest producer is the FETCH node of the same instruction; on the other hand, since the data of the earlier FETCH nodes have been consumed by preceding DECODE nodes, the latest FETCH node is also the producer of the first available data at the time when DECODE is being processed. As a result, a producer-consumer relation between the the latest FETCH node and the DECODE node is constructed *automatically*. In contrast, this relation in [10] is constructed based on human understanding of the pipeline model.

Once the graph relations have been constructed, the estimation algorithm, an extension from [10], can be applied to the execution graph to obtain the worst case estimate for the sequence of instructions.

V. Case Study

In this section, we present the targeting from an ADL toolkit, EXPRESSION [4], to WCET analysis for a realistic processor (acesMIPS).

A. ADL Specification

The acesMIPS processor, shown in Fig.4, is similar to MIPS R4000. The ADL specification captures architectural components and their connectivity as a netlist. Some snippets of the description are given in Fig.5. The pipeline section (Fig.5a) specifies the overall pipeline structure, where FETCH, DECODE and WB are units. A unit is an instance of a unit class, e.g., DECODE is from DecodeUnit, and ALU1_READ is from OpReadUnit. The parameters of DECODE and ALU1_READ are described in the ADL (Fig.5d and Fig.5e). In contrast, the properties of unit classes are described in C++, e.g., DecodeUnit is defined in DerivedUnit.h (Fig.5c). In addition to concrete units, the top-level pipeline description also contains a sub-pipeline, READ_EXECUTE. The subsequent lines in Fig.5a indicates that this sub-pipeline is a collection of alternative two-

TABLE I
ADL-WCET component mapping

ADL components	WCET Components	
	Input side	Output side
FetchUnit	Array	FIFO
DecodeUnit	FIFO	FIFO
OpReadUnit	Array/FIFO	FIFO
ExecuteUnit	FIFO	FIFO
LoadStoreUnit	Array/FIFO	Array/FIFO
WritebackUnit	FIFO	Array

stage paths. This structure enables the ADL to describe non-trivial processor models.

B. Targeting acesMIPS for WCET Analysis

Targeting acesMIPS architecture to WCET analysis mainly includes the three tasks described in Section IV(B) . Although building the component map table is not automated, it does not need much effort for acesMIPS since it contains only a small number of component classes, which are also very simple ones. Table I presents this the map table.

There is one ADL component not mapped to the WCET component library – BranchUnit. Usually, for a component to be retargetable, either (1) its processing element has very simple timing behavior (i.e., the component is characterized by its storage elements); or (2) the processing element has complex behavior, but it does not change wildly across different processor models. Unfortunately, the branch processing unit does not fall into either category. It is well known that branch processing, including prediction and recovery, often spans across multiple pipeline stages and its behavior is highly architecture dependent. Thus, it is difficult to write a branch processing unit retargetable to a variety of processor models, either for ADL or for WCET.

From above discussion, we can see that a hard-to-retarget component in our framework is hard to retarget in ADL description in the first place, so what we can do is to read the code

TABLE II
SIMULATION AND WCET ESTIMATION RESULTS

Benchmark	Simulation	Estimation	Ratio
fdct	10310	12301	1.19
fft	2937925	3880257	1.32
isort	149057	175724	1.18
matmul	57431	72150	1.26
matsum	442556	482570	1.09

written manually in ADL and write its WCET counterpart.

C. Experimental results

In this part, we present the estimated WCET (an upper bound on actual WCET) and compare it with the one from acesMIPS simulator (a lower bound to actual WCET).

The results in Table II show that the estimation numbers are close to the simulated ones. From the WCET framework presented in Section III, we know that there are two sources of overestimation: program path analysis and basic block timing analysis. Program path analysis is the same as in [10]. For basic block timing analysis, the automatically constructed execution graphs capture all dependences and contentions that are captured in [10]. So the retargetable framework introduces no more accuracy loss than the manual method in [10].

This framework was implemented by one person (the first author), and the amount of work can be viewed as two parts: developing the retargetable framework and retargeting it to acesMIPS. The first part includes (1) automatic construction of execution graphs and generalization of the WCET analysis in [10]; (2) the construction of the WCET component library; (3) the EXPRESSION-WCET interface, e.g., parsing of ADL description. This part was done in two months, and it is a one-time effort. The second part includes (1) ADL-WCET component mapping that results in a mapping shown by Table I; (2) Handcrafted code for components not mapped (BranchUnit); (3) Simulation, estimation and result analysis. This part was done in half month. Note that acesMIPS description is provided by the EXPRESSION group. In general, writing a processor model, no matter what tool is used, is a considerable amount of work. As we are only concerned with WCET analysis, writing a processor model is not considered as part of the retargeting effort.

VI. CONCLUSIONS

Two factors contribute to the challenge of WCET analysis: the complexities of the hardware, and the intolerance of any underestimation, which means techniques for average performance evaluation cannot be applied here. As a result, it is very important to automate the process of hardware modeling. In this paper, we have proposed a solution by using an architecture description language (ADL), which are gaining acceptance in describing embedded processors.

The retargetable framework in based on the key observation that the behaviors of pipeline processing elements are often dictated by their input/output storage elements, which can be generalized into a few classes convenient for WCET analysis. The effectiveness of the proposed framework has been proved with a case study on a MIPS-like processor written in EXPRESSION, an ADL for design space exploration of embedded processors.

In the future, we will investigate better ways of interfacing non-pipeline components with the pipeline to further reduce human intervention in the analyzer.

ACKNOWLEDGMENTS

This work was partially supported by University Research Council (URC) project R252-000-171-112 from National University of Singapore (NUS).

REFERENCES

[1] K. Chen, S. Malik, and D.I. August. Retargatable static software timing analysis. In *IEEE/ACM Intl. Symp. on System Synthesis (ISSS)*, 2001.

[2] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Real time Systems*, May 2000.

[3] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, 2002.

[4] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *DATE*, 1999. `http://www.ics.uci.edu/~express/`.

[5] M.G. Harmon, T.P. Baker, and D.B. Whalley. A retargetable technique for predicting execution time of code segments. *Real-Time Systems*, 1994.

[6] R. Heckmann et al. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), July 2003.

[7] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Static Analysis Symposium (SAS)*, 2002.

[8] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *DAC*, 2003.

[9] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *IEEE Real-Time Systems Symposium*, 2004.

[10] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Journal of Real-Time Systems*, 34(3), 2006.

[11] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM ToDAES*, 4(3), 1999.

[12] Prabhat Mishra and Nikil Dutt. *Functional Verification of Programmable Embedded Architectures: A Top-Down Approach*. Springer, 2005.

[13] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-time Systems*, 1(2), 1989.

[14] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An efficient retargetable framework for instruction-set simulation. In *CODES+ISSS*, 2003.

[15] S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *DAC*, 1998.

[16] O. Schliebusch et al. RTL processor synthesis for architecture exploration and implementation. In *DATE*, 2004.

[17] A.C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 1(2), 1989.

[18] W. Qin, S. Rajagopalan and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES*, pages 47–56, 2004.

[19] R. Wilhelm and C. Ferdinand. On predicting data cache behavior for real-time systems. In *LCTES*, 1998.

[20] S. Wilhelm. Efficient analysis of pipeline models for wcet computation. In *WCET Workshop*, 2005.