TOWARDS PREDICATED WCET ANALYSIS

Amine Marref, Guillem Bernat[1]

Abstract
In this paper, we propose the use of constraint logic programming as a way of modeling context-sensitive execution-times of program segments. The context-sensitive constraints are collected automatically through static analysis or measurements. We achieve considerable tightness in comparison to traditional calculation methods that exceeded 20% *in some cases during evaluation. The use of constraint-logic programming in our calculations proves to be the right choice when compared to the exponential behaviour recorded by the use of integer linear-programming.*

1. Introduction

WCET analysis have been explored for about two decades and can be divided into three categories: end-to-end testing, static analysis (SA), and measurement-based analysis (MBA) [6]. SA and MBA finds the WCET of a program as follows: (a) decomposing the program into segments, (b) finding the execution times of these segments, and (c) combining these execution times using a calculation technique: tree-based [5], path-based [8], or implicit path-enumeration (IPET) [9, 14]. Path-based methods suffer from exponential complexity and tree-based methods cannot model all types of program-flow, leaving IPET as the preferred choice for calculation because of the ease of expressing flow dependencies and the availability of efficient *integer linear-programming* (ILP) [1] solvers.

Current calculation techniques struggle to cope with variations in execution times of program segments caused by modern-hardware speed-up features because of the complexity resulting from modeling all these timing variations. This motivates the use of a more powerful calculation technique which copes with execution time variations and yields tighter, more context-sensitive WCET estimations.

We proceed by identifying the necessary conditions leading to the observation of different execution times of program segments. These conditions are expressed as implications which by definition are disjunctions (if a and b are predicates than $(a \Rightarrow b) \equiv (\neg a \vee b)$). There can be many segments which have multiple execution times, and each execution time of the segment is caused by one or more segments that previously executed. This makes the total number of constraints to handle considerably large.

In the current work, we use *constraint-logic programming* (CLP) [2] in order to express the constraints governing the execution flow and times of the segments in the program. All constraints including implications/disjunctions can be encompassed in the same model using CLP and with no model expansion. These two features make CLP solve an IPET model within seconds, which is otherwise solved using ILP in hours because of model duplication (ILP handles disjunction through model duplication). CLP also enables the integration of execution-time analysis of many hardware components (Section 6), an issue that has never been properly resolved (Section 2).

[1]Department of Computer Science, University of York, Heslington, YO105DD, UK

ECRTS 2008
8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2008/1667

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 establishes the required terminology. Section 4 explains the deriving of constraints required for CLP calculation by either static analysis or through measurements. Section 5 explains how the constraints are expressed in CLP and highlights the problems encountered when trying to express the constraints in our work using ILP. Section 6 shows an example where constraints derived from many hardware components analysis are integrated together. Section 7 shows the tightness in WCET estimations we obtained during the evaluation of our context-sensitive calculation. Finally, Section 8 summarizes the important results and sets aims for future work.

2. Related Work

WCET analysis using context-sensitive IPET is not novel, the idea has been around for a long time. What is common in the literature is that execution-context could not be fully exploited because of the exponential complexity that usually accompanies the process. In addition, the focus has always been to achieve context sensitivity in the execution counts of segments rather than the execution times due to the fact that ILP is used to compute the solution which is by definition linear (i.e. either execution counts or execution times can vary but not both).

There has been much work on WCET calculation using IPET starting in [9, 14] where the execution times are constant. In later work e.g. [10, 4], ILP was used to represent more execution history by modeling caches, pipelines, branch predictors, and speculative execution. The objective function is generally augmented by execution-time gains/penalties resulting from the use of the hardware component being analysed. These objective functions have not been integrated together because of their complexity. Each segment has two execution times at most.

In a very recent work [16], timing variability of basic blocks with respect to pipelines has been analysed where there can be multiple execution times of a basic block depending on the subpath previously traversed, then ILP is used to calculate the overall WCET.

By and large, the work in [7] is the most related to ours. In [7] WCET calculation is performed using the notion of scopes to provide context-sensitivity mainly through constraining execution counts. At the low-level, the execution-times of segments are expressed in a scenario-based fashion where multiple execution times are allowed in theory. The integration in the ILP model requires bounds on different execution times of a basic block to be known a priori which as we shall see (Section 5.2) still generates pessimism and are very hard to derive. The work in [7] assumes that the constraints are provided by a user, so although the calculation is context-sensitive, its practical value is limited to when such constraints are available. The low-level constraints such as "block B_1 has got execution time 100 at most six times" are very unlikely to be provided by the user and must be derived automatically which is hard. This leads to the fact that the evaluation incorporated only the pipeline effects of a block over its immediate predecessor i.e. a block has at most two execution times.

Our work is different firstly in the sense that it allows all conditional execution-times to be expressed easily so there is no need for akwardly written ILP objective functions. The constraints governing the observation of execution times of a block are determined automatically without the need to put bounds on the number of such observations which are then computed exactly using CLP. Secondly, all constraints with respect to all hardware components are integratable together by simple conjunctions. Thirdly, solving these constraints using CLP takes seconds while it can take hours using ILP. Finally, the calculation is path-sensitive and introduces no pessimism as we shall demonstrate in Section 5.2.

3. Definitions

The core unit of the proposed calculation method is the basic block which is a contiguous sequence of instructions where the first instruction is jumped to (or is the first instruction of the program) and the last instruction is jumped from (or is the last instruction of the program) [11]. Each basic block B_i ($i \in [1..n]$) is associated an execution count x_i and an execution time c_i. When B_i has θ_i execution times, these will represented as c_i^j where $j \in [1..\theta_i]$. We also define $wcet_i$, $bcet_i$ to be the WCET and BCET of block B_i respectively.

The program under-analysis is represented by a control-flow graph (CFG) which is defined as a tuple (V, E). V is the set of vertices, in this case the basic blocks in the program, $|V| = n$. E is the set of edges, which in this case are the transitions between the basic blocks in the program.

In order to express conditional execution-time observation, we use the operator '$/$'. For example $c_{j/i} = \alpha$ means that $c_j = \alpha$ iff B_i is executed. A block B_i is the predecessor of a block B_j if there is sequence of one or more edges from B_i to B_j not containing a back-edge [11].

4. Deriving the Constraints

We use *predicated WCET analysis* which we define as performing WCET analysis by considering all different execution times of a program segment and expressing them as the outcomes of executing some other segments in the past. There is therefore the need to (a) identify the different execution times of a program segment and (b) identify the -previously executed- program segments that cause these execution times. Without loss of generality, we use the basic blocks as our program segments.

4.1. Constraints Using SA

The number of different execution times of a basic block varies with the complexity of the architecture where it runs. A complex architecture causes a basic block to exhibit a large number of different execution times. So far, we can express execution dependency constraints with respect to instruction caches (icaches), data caches (dcaches), static branch predictors, and pipelines. To maintain clarity, we will only address the derivation of icache constraints.

The analysis of the icache deals with deriving execution time dependencies between blocks in the CFG by exploiting basic-block layout in main memory[2]. We start by finding the WCET and BCET of all basic blocks. If B_i shares a program line with B_j, then the effect is expressed using the constraint $x_i > 0 \Rightarrow c_j = c_{j/i}$. We determine how many program lines belonging to B_j are loaded by the execution of B_i, let this be α lines. Block B_j acquires a new execution time $c_{j/i} := wcet_j - \alpha \times (i_m - i_h)$ where i_m is the icache miss latency, and i_h is the icache hit latency. If B_i displaces a program line used by B_j in a loop, then similarly $c_{j/i} := bcet_j + \beta \times (i_m - i_h)$.

The analysis is easily automated. For each block B_i, we find the blocks that share program lines with it, and the blocks that conflict with it in some icache blocks. The start and end addresses of the basic blocks, together with knowledge about the icache architecture enable the block execution-time dependency-analysis with respect to the icache.

[2]The CFG is constructed from the disassembled binary file of the program and hence basic block start and end addresses - in main memory - are available.

Studying cache conflicts is not novel in this work as it has been used in SA in the past [12]. The novelty here is in using SA to derive new execution times and link these execution times to past execution. Müeller [12] performed a complete analysis on instruction caches where icache accesses are identified as being hits, misses or unknowns. Since the analysis must be safe, unknowns are considered as being misses. This can be a great potential of pessimism in the evaluated WCET. In Figure 2(a) - ignoring the constraints - the returned WCET is 3200 assuming $c_3 = 70$.

4.2. Constraints Using Traces

Execution-trace analysis can also be used to derive the constraints of the CLP problem. An execution trace is a time-stamped execution of the program which can be obtained using a tracing method [13]. It contains all instructions executed during a particular run of the program with timing information. The execution trace can be exploited to derive constraints on the execution-counts of program segments or constraints on their execution-times.

Conditional execution times can be learnt from traces where a particular execution time of a block B_1 is recorded whenever B_2 is executed. The quality of the generated traces affects the correctness of the derived constraints. If traces are generated using full-path coverage, it is guaranteed that the timing constraints are learnt exactly. However, path coverage is impractical, so a less costly coverage metric must be employed. Unfortunately, functional testing coverage metrics are not adequate for our task as they do not consider the temporal properties of the program, and hence there is a need for new coverage metrics. We are currently exploring ways of generating appropriate test vectors that help obtain maximum variability in block execution times and executed paths using genetic algorithms.

5. Modeling the Constraints

In the last section we explain how conditional execution-times are expressed using implications. In this section we explain how CLP is used to model these constraints. In order to see the benefits of using CLP in our work, a comparison against ILP is made to illustrate the constraints that can be expressed better using CLP. In literature [9, 14], the constraints used in ILP are *flow* constraints. We use flow constraints and introduce *time constraints*.

5.1. Flow Constraints

These constraints express the rules governing the execution flow and dependencies in a program, these are divided into structural and functional constraints. Structural constraints preserve the execution flow of the program, and functional constraints describe aspects of program-execution behaviour. For a formal description, see [9, 14]. For instance, for any two blocks B_i and B_j, if we want to express that they are on the same path where B_j is inside a loop, the constraint $(x_i > 0 \wedge x_j > 0) \vee (x_i = 0 \wedge x_j = 0)$ is used in the CLP model. When the two blocks are outside any loop, the constraint $x_i = x_j$ is enough to express same-path relation. Mutual exclusion is represented similarly.

In this paper, the only type of flow constraints that is included in the CLP model are structural constraints. We do not detect functional constraints such as infeasible paths, so the model does not incorporate them. However, they can be added if available.

5.2. Time Constraints

These constraints describe the necessary conditions - expressed in terms of execution flow - that must hold to give rise to a particular execution time of some basic block. Given a basic block B_i with θ_i execution times $c_i^1, c_i^2, ..., c_i^\theta$, B_i is affected by a set Ψ_i of σ_i blocks $B_1, B_2, ..., B_{\sigma_i}$. In general $i \notin [1..\sigma_i]$, but in some special cases where the block size is larger than the icache size or when it accesses a variable whose size is larger than the dcache $i \in [1..\sigma_i]$. Every block B_k, $k \in [1..\sigma_i]$ can either execute or not execute, so there is a total of 2^{σ_i} different effects on block B_i. These effects are best visualized by imagining a truth table of θ_i variables where a 0 means block not executing and 1 means block executing. The relation $2^{\sigma_i} \geq \theta_i$ must hold because every execution time of a block B_i must be related to previous execution history. When $2^{\sigma_i} > \theta_i$, there will be some effects of the blocks in Ψ_i that are either equivalent (map to the same execution time) or impossible (the corresponding combination of blocks is not possible). Notice that in the architecture we consider, θ_i is usually small. Blocks B_i with large size can have a considerable θ_i.

Impossible effects can be ruled-out before passing the time constraints to the solver, or they can (eventually) be eliminated by the solver. Equivalent effects can be simplified using boolean algebra techniques and then passed to the solver. Obviously, the degree of simplification will be different for every equivalence class of time constraints.

Next we need to generate the conditional execution-time relations. Conditional execution times of B_i are expressed using

$$(x_1 \odot 0 \wedge x_2 \odot 0 \wedge ... \wedge x_{\sigma_i} \odot 0 \Rightarrow c_i = c_i^j) \tag{1}$$

where each $\odot$ stands for greater than ($>$) xor equal ($=$) (the $\odot$ can have a different instantiation in each occurrence in the same time constraint). The time c_i^j is the execution time observed for a given instantiation of the operators $\odot$ e.g. $(x_1 > 0 \wedge x_2 > 0 \wedge ... \wedge x_{\sigma_i} > 0 \Rightarrow c_i = 100)$.

Adding more constraints to the constraint model generally helps prune the search. The potential large number of time constraints is expected to speed-up the constraint search. For example, assume two blocks B_1, B_2 affecting the execution time of a third block B_3 in the following way:

$$\begin{array}{l} (x_1 = 0 \wedge x_2 = 0 \Rightarrow c_3 = 1) \\ \wedge(x_1 = 0 \wedge x_2 > 0 \Rightarrow c_3 = 2) \\ \wedge(x_1 > 0 \wedge x_2 = 0 \Rightarrow c_3 = 3) \\ \wedge(x_1 > 0 \wedge x_2 > 0 \Rightarrow c_3 = 4) \end{array} \tag{2}$$

The search space is

$$(x_1, x_2, c_3) \in \{(\{0\}, \{0\}, \{1\}), (\{0\}, \mathbb{Z}^{+*}, \{2\}), (\mathbb{Z}^{+*}, \{0\}, \{3\}), (\mathbb{Z}^{+*}, \mathbb{Z}^{+*}, \{4\})\} \tag{3}$$

In the absence of these constraints, the search space is:

$$(x_1, x_2, c_3) = (\mathbb{Z}^+, \mathbb{Z}^+, \{1, 2, 3, 4\}) \tag{4}$$

The size of the search space in Formula 3 is $(|\mathbb{Z}^{+*}|^2 + 2 \times |\mathbb{Z}^{+*}| + 1)$. The size of the search space in Formula 4 is $(4 \times |\mathbb{Z}^+|^2)$. As can be seen, the time constraints partition the (non-linear) search space.

It is still possible to express conditional execution times in ILP at the cost of great complexity. This can be achieved through model duplication (ILP1) or bounds on execution times (ILP2).

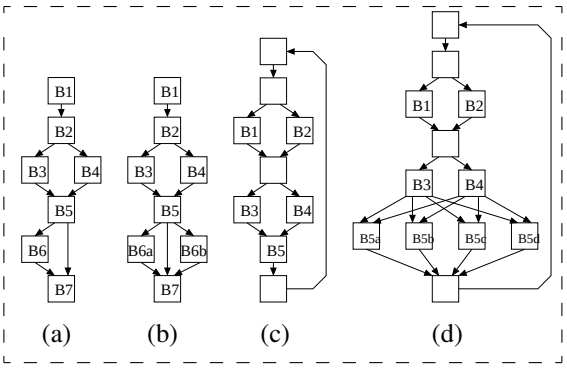

Figure 1: Time constraints for ILP

ILP1. Consider Figure 1(a) where $c_{6/3} = 7$ and $c_{6/4} = 10$. The basic ILP formulation of the problem when c_6 is constant is to maximize the sum $\sum_{i=1}^{7} c_i \times x_i$. When c_6 is not constant, the term $c_6 \times x_6$ needs to be expanded further. This is done by duplicating B_6 as is shown in Figure 1(b) and adding mutual-exclusive path information to the model.

In Figure 1(b), B_6 with execution times $\{7, 10\}$ is expanded to B_{6a} with execution time $c_{6a} = 7$ and B_{6b} with execution time $c_{6b} = 10$. Since $c_6 = c_{6a} = 7$ is observed only when B_3 is executed, we can state that B_{6a} is mutually exclusive with B_4. The same argument is made for B_{6b} and B_3. The updated ILP formulation becomes $(c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 + c_5x_5 + c_{6a}x_{6a} + c_{6b}x_{6b} + c_7x_7)$ with the additional constraints that express mutual exclusivity. The ILP problem needs to be solved for each set of mutual exclusive paths, then the best solution is taken. The number of model copies to solve grows exponentially with the number of nodes that have multiple execution times and the number of different times they have (e.g. Figures 1(c), 1(d)).

ILP2. The other way to express conditional execution-times is to impose bounds on the number of times each single execution time is observed. This allows all constraints to be solved by a single run of the model and with expanding only the blocks in question. However, this only works provided the bounds on the observation of different execution times are available which is very hard to determine statically. In addition, the returned WCET will not be as accurate as the WCET returned by CLP or ILP1. The reason for this is that there is no path information in ILP2 compared to CLP, ILP1.

6. Example of Integration

Figure 3 shows (a) a program written in pseudo-assembly, (d) its CFG, (b) its block memory layout, and (c) the referenced variables placement in the dcache. We are interested in analysing the execution time of B_4 which has the value $wcet_4$ in its worst case. This is equivalent to performing 3 icache misses, 2 dcache misses, and starting execution from a flushed pipeline. Block B_4 is reached from three blocks: B_1, B_2, B_3 where no block in these three blocks is a predecessor of another one. Assume all blocks are outside any loop.

If B_4 is executed after B_1, it gains nothing in execution time with regards to icache because B_1 loads program lines PL_1 and PL_2 neither of which is used by B_4. Block B_4 however gains in dcache execution by $1 \times (d_m - d_h)$ because B_1 loads data line DL_1 which is used by B_4. Finally, B_4 gains

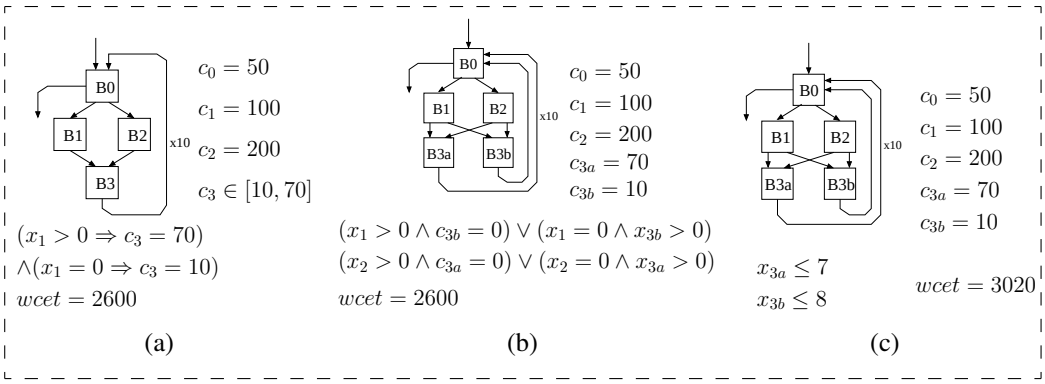

Figure 2: The time constraints and the corresponding WCET using CLP (a), ILP1(b), and ILP2(c). In (a) the nodes not duplicated, conditional execution times are added. In (b) and (c), the nodes with variable execution times are duplicated. In (b), mutual exclusion constraints are added. In (c), bounds on the execution counts of nodes with variable execution times are added.

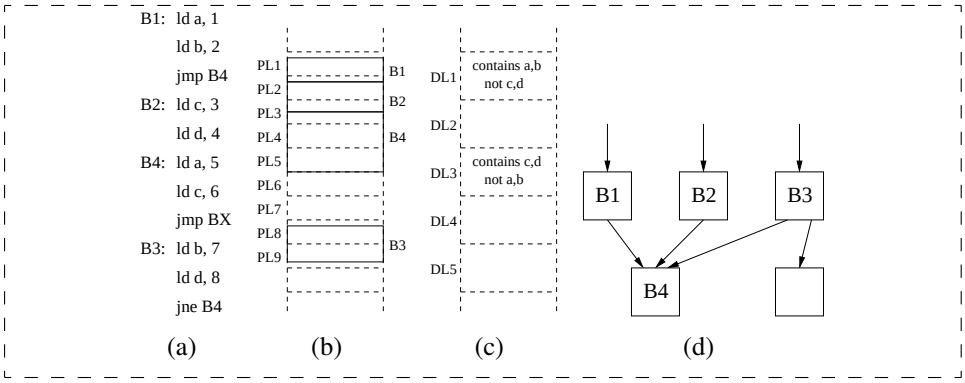

Figure 3: The disassembly code, memory layout, dcache content and a CFG window of a block B_4 affected by blocks B_1, B_2, and B_3

g_1 cycles because of the pipelined execution of $B_1; B_4$ (no misprediction can occur as the jump is unconditional).

If B_4 is executed after B_2, it gains $1 \times (i_m - i_h)$ with regards to icache because B_2 loads program line PL_3 which is used by B_4. Block B_4 gains in dcache execution by $1 \times (d_m - d_h)$ because B_2 loads data line DL_2 which is used by B_4. Finally, B_4 gains g_2 cycles because of the pipeline execution of $B_1; B_4$. Here assume $g_2 > g_1$.

If B_4 is executed after B_3, it gains nothing in execution time with regards to icache because B_1 loads no program line that is used by B_4. Block B_4 gains in dcache execution by $2 \times (d_m - d_h)$ because B_3 loads data lines DL_1, DL_2 which are used by B_4. Finally, B_4 gains g_3 cycles because of the pipeline execution of $B_3; B_4$. Here assume $g_2 > g_3 > g_1$.

The execution time of B_4 is captured by the constraint:

$$\begin{aligned}
&(x_1 > 0 \Rightarrow c_4 = wcet_4 - (d_m - d_h) - g_1) \wedge \\
&(x_2 > 0 \Rightarrow c_4 = wcet_4 - (i_m - i_h) - (d_m - d_h) - g_2) \wedge \\
&(x_3 > 0 \Rightarrow c_4 = wcet_4 - 2 \times (d_m - d_h) - g_3)
\end{aligned} \tag{5}$$

7. Evaluation and Results

We obtain the execution times using Simplescalar [3]. We use pollution techniques to force the WCET, BCET of each basic block in the program and compute this WCET, BCET by means of measurements. The hardware used comprises a CPU with a single-issue in-order pipeline, icache L1, dcache L1, and a static branch predictor. First, we analyse dependencies between basic blocks with respect to the icache as we discuss in Section 4.1 (the process is automatic). Then we derive the corresponding time constraints, and solve the constraints using ECL^iPS^e, a constraint logic programming engine [2].

The objective of the evaluation is to show (a) that PWA yields tighter WCET estimations in comparison with HMU, and (b) show that the solution time to solve the CLP model is affordable.

We compare the tightness of the WCET values obtained using our Predicated WCET Analysis (PWA) with a method that uses Hit, Miss, first-hit, first-miss, Unknown analysis (HMU) [12]. An HMU icache analysis method quantifies the number of icache hits and misses per basic block. When the icache access is guaranteed to be a hit or a miss, it is classified accordingly. When the icache access is not guaranteed to be a hit or a miss (i.e. unknown), it is classified as a miss to achieve safety. Our analysis method puts more context-sensitivity in the icache analysis by stating the condition under which the icache access (considered a miss by HMU) will hit or miss.

We have tested our tool on some WCET benchmarks available from [15]. Table 1 shows the execution times obtained using PWA and HMU on a representative[3] subset of the benchmarks. The icache has a size of $1k$ bytes. As can be seen, considerable tightness has been achieved in WCET for the first three programs (*select, fdct, fir*) which can be explained by the large number of constraints -relative to the number of blocks- which allows less pessimism during the constraint search.

The fourth program (*lms*) -although having the largest relative number of constraints in the table- does not have the best WCET tightness using PWA. This is due to the nature of the constraints involved in calculation. In our implementation, as a temporary solution to manage the large number of constraints that a particular block can have, we decide that each basic block can be constrained by at most five blocks. When a block is constrained by more than five blocks, its WCET is used during calculation. The program *lms* has got a big loop which consumes more than half the number of its blocks (*73*), which leads to many icache conflicts given the used icache configuration.

The last three programs (*cnt, bsort,* and *ns*) scored very small tightness. When these programs are run on an icache with smaller size, they generate more constraints and score greater tightness.

The CLP solving time during evaluation (including some other programs) did not exceed a few seconds. The solving process in ILP is usually instantaneous for one run but then becomes exponential

[3]Representative in terms of tightness i.e. the tightness scored with other programs from the benchmarks has more or less one of the values shown in Table 1.

Table 1: WCETs of benchmark programs using PWA and HMU

#	program	blocks	implications	wcet		gain
				HMU	PWA	
1	select	40	27	558627	432803	22.6%
2	fdct	12	6	77759	66975	15%
3	fir	17	4	87822	81742	7%
4	lms	134	86	747776	724752	4.3%
5	cnt	36	2	94672	92912	1.9%
6	bsort	20	4	58179	57539	1.2%
7	ns	22	5	892708	888148	0.6%

when running all duplications. We use *lp_solve* to solve each of the (linear) disjunctive ILP instances (ILP1). Each instance is solved in few micro seconds. Using a more powerful ILP solver such as *CPLEX* might cut down the time required to solve one instance of the disjunctive ILP. However, this will only mean that (few) more time constraints can be tolerated. The exponential behaviour is still present.

If for instance, the model has n time constraints and *lp_solve* takes α units to solve each instance of ILP1; the number n' of time constraints that can be solved using *CPLEX* in the same amount of time is $n' = n - ln(2) \times ln(\alpha/\beta)$ where β is the time taken by *CPLEX* to solve one instance. So if *CPLEX* was a million times faster than *lp_solve* ($\alpha = 10^6 \times \beta$), *CPLEX* can solve the same model with extra 10 time constraints in the same amount of time. The models in our case were solved in few micro seconds, if the solver was a million times faster or more, they would be solved in few pico seconds or less which is doable only by super computers.

8. Conclusions and Future Work

In this paper we have proposed the use of constraint-logic programming (CLP) to compute tight values of WCET by using constraints derived through execution-time dependency-analysis. In this work we have considered icache constraints only and we concluded that CLP is superior to integer linear-programming (ILP) whenever there is a reasonable number of execution-time dependencies. The choice of whether or not to use predicated WCET analysis (PWA) and CLP is dictated by the nature of the program. If execution-time dependency-analysis reveals lots of constraints, it is worth using PWA and CLP because considerable tightness may be achieved. In a future work, we will show how constraints from other hardware components are derived. We are currently investigating how to prove the safety of constraints derived using tracing. We are also working on improving the calculation method so that constraints are solved more efficiently.

References

[1] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[2] K.R. Apt and M. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[3] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, 1997.

[4] C. Burguiere and C. Rochange. A contribution to branch prediction modeling in WCET analysis. In *Proceed. of the conf. on Design, Automation and Test in Europe*, pages 612–617, Washington, USA, 2005. IEEE Computer Society.

[5] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems, Special issue on worst-case execution time analysis*, 18(2):249–274, April 2000.

[6] J.F Deverge and I. Puaut. Safe measurement-based WCET estimation. In *Proceedings of the 5th International Workshop on Worst Case Execution Time Analysis*, pages 13–16, Palma de Mallorca, Spain, July 2005.

[7] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, August 2003.

[8] C.A. Healy, R.D. Arnold, F. Müeller, M.G. Harmon, and D.B. Walley. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, 1999.

[9] Y.T. Steven Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[10] T. Mitra and A. Roychoudhury. A framework to model branch prediction for worst case execution time analysis. In *Proceedings of the 2nd Workshop on WCET Analysis*, October 2002.

[11] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[12] F. Müeller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):217–247, 2000.

[13] Stefan M. Petters. Comparison of trace generation methods for measurement based WCET analysis. In *Proceedings of the 3rd International workshop on worst-case execution time (WCET) analysis*, pages 75–78, July 2003.

[14] P. Puschner and A.V. Schedl. Computing maximum task execution times - A graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

[15] Mälardalen WCET research group. Wcet project/benchmarks. *http://www.mrtc.mdh.se/ projects/wcet/benchmarks.html*, January 2008.

[16] C. Rochange and P. Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Transactions on High-Performance Embedded Architecture and Compilation*, 2(3):109–128, 2007.