Worst Case Execution Time Analysis for Modern Hardware Architectures

Greger Ottosson
Computing Science Dept.
Uppsala University
Uppsala, Sweden
e-mail: greger@csd.uu.se

Mikael Sjödin
Dept. of Computer Systems
Uppsala University
Uppsala, Sweden
e-mail: mic@docs.uu.se

June 15, 1997

ABSTRACT

Knowing the worst case execution times (WCETs) for programs are crucial for the design and verification of real-time systems. Modern hardware architectures utilize pipelined execution and cache memory for improved performance. We extend an existing execution time analysis technique, the Implicit Path Enumeration Technique (IPET), to consider these and other modern hardware architecture features.

We extend IPET in two stages. First, we annotate the control flow graph of the program with variables representing the history of execution, thus allowing the state of architectural entities, such as cache and pipeline, to be determined before each basic block. Secondly, we model the architectural entities with constraints. The result is an equation which contains a complete model of how the program will execute on the modeled architecture.

This novel idea provides a straightforward and flexible way of incorporating the behavior of various modern hardware architecture features into WCET analysis.

1 Introduction

One of the key questions when designing a real-time system is resource allocation. In solving this problem it is vital to have a good estimation of the resource requirements of the different components in the system. For software components, execution time is the most important resource. Determining the *Worst Case Execution Time* (WCET) for software components is a prerequisite for most real-time analysis, e.g. schedulability analysis.

Background: Traditionally, cache behavior has been deemed too complex to analyze for real-time systems. But recently several methods for schedulability analysis of cached architectures have emerged, e.g. [BN94, BMSO$^+$96, LHS$^+$96]. Similarly, much of previous work on WCET estimation, e.g. [PK89, PS91, Gus94, CBW94, PS95, LM95], has not considered cached architectures. Recently researchers have proposed methods which allow tighter estimation of WCET in cached systems, e.g. [AMWH94, LL94, LBJ$^+$95, KMH96, LMW96]. Arnold et.al. [AMWH94] consider only instruction caches which partly limit the applicability of their analysis. Lim et.al. [LBJ$^+$95] extend the original timing schemas, proposed by Puschner and Koza [PK89], to handle pipelined and cached architectures. Kim, Min and Ha [KMH96] further refine this method with a better model for data caches. The result is that the actual execution of the object code is modeled very well. However, the timing schema approach lacks the possibility to capture semantical dependencies in the code. For instance, it is very hard to express that the number of iterations in an inner loop is dependent on the loop-index of an outer loop, or that two sub-paths at different locations in the code are mutually exclusive. The method in [LL94] suffers from similar problems.

In recent work Li, Malik and Wolfe [LMW96] revise the work in [LM95] and the *Implicit Path Enumeration Technique* (IPET) is extended to allow pipeline execution and cache memory to be modeled with linear constraints. In IPET [LM95, PS95] all the possible execution paths of a program is described by placing constraints on, and relations between, the number of executions of different parts of the program. An execution time expression is formulated by multiplying the time to execute each part of the program with the number of times the part is executed. The WCET is found by maximizing this expression under the constraints, using linear programming (see e.g. [NRT89]).

IPET allows semantical dependencies to be expressed as constraints on the control flow graph of the program. This possibility is important to avoid execution time for *infeasible paths* to be part of the WCET estimation. Thus, IPET is very suitable to combine with advanced semantical analysis at the source code level (since the information derived in the semantical analysis can be

expressed as constraints in the flow graph).

Our Contribution: We present a method for estimating the WCET for programs running at modern hardware architectures. Our method is based on IPET, but our approach is quite different from the work in [LMW96]. Li, Malik and Wolfe [LMW96] use a framework specialized for cache modeling, while we present a general framework for representing the history of the execution and micro-architectural modeling. This allows us to express pipeline and cache effects in the same general framework. It is also possible for us to express other micro-architectural aspects, such as instruction level parallelism or EDO-RAM (EDO-RAM speed up accesses to consecutive memory addresses), within the same framework.

We extend IPET in two stages. First, we annotate the control flow graph of the program with variables representing the history of execution, thus allowing the state of architectural entities, such as cache and pipeline, to be determined before each basic block. Secondly, we model the architectural entities with constraints. The result is an equation which contains a complete model of how the program will execute on the modeled architecture. Optimizing this equation with respect to execution time, using constraint satisfaction methods, yields the WCET of the program.

Paper Outline: Section 2 recapitulates the foundations of IPET as published before and describes our extensions to the method. Sections 3 and 4 describe how the pipeline and cache effects are modeled. Section 5 introduces Constraint Satisfaction techniques which are used to find the WCET. In section 6 we give an example of how our method can predict the worst case cache behavior and we conclude our paper in section 7.

2 Calculating the WCET

We begin our description by recapitulating the work described in [PS95] (section 2.1). We then describe our extension to that method (section 2.2), which allows us to consider cache and pipeline effects.

2.1 The Basic Method

A program is described by its control flow graph G. This is a directed graph where each *basic block* is represented by an edge e_i. Figure 1(a) shows an example of a program and 1(b) its corresponding control graph[1]. Each edge, e_i, is associated with (1) a worst case execution time, c_i, computed from the object code of the basic block, and (2) the maximum number of executions of edge e_i in any execution of the program, denoted λ_i. λ_i can be determined from the maximum number of iteration in loops, either by semantic analysis (e.g. [CBW94]) or by source code annotations (e.g. [PK89]). In figure 1(c) the maximum number of executions of each edge for our sample program is shown.

For an execution path P (i.e. P represents one specific execution) each basic block e_i is executed a number of times, denoted by x_i^P. The total time spent in edge e_i is expressed by $t_i^P = x_i^P * c_i$. Thus, the total execution time of the program (ET^P) is the sum of the times spent in each edge on path P, i.e.

$$ET^P = \sum_{i \in \text{edges}(G)} t_i^P = \sum_{i \in \text{edges}(G)} x_i^P * c_i \qquad (1)$$

To find the Worst Case Execution Time it is necessary to find the maximum value of ET^P of any execution of the program. Let χ^G denote the set of possible executions of control graph G, then

$$\text{WCET} = \max_{P \in \chi^G} ET^P = \max_{P \in \chi^G} \sum_{i \in \text{edges}(G)} x_i^P * c_i \qquad (2)$$

For notational convenience we ignore the superscript ^{P} for x_i^P for the rest of this paper, when unambiguous.

Clearly, the x_is cannot be assigned arbitrary values. For instance, they are all non-negative integers, and they are bounded by their respective λ_i. Formally

$$x_i \in \mathbb{N}_0, \text{ and } x_i \leq \lambda_i \qquad (3)$$

However, these restrictions on the x_is are not enough to characterize the possible executions of the program. In [LM95, PS95], it is described how constraints are placed on the x_is so that they can only take values that represent valid executions. These constraints are all linear relations between the different x_is.

The most fundamental constraints express the preservation of flow. That is, for each node $n \in G$, it must hold that the sum of the incoming x_is is equal to the sum of the outgoing x_js. Formally[2]

$$\sum_{i \in \text{in}(n)} x_i = \sum_{j \in \text{out}(n)} x_j \quad (\forall n \in \text{nodes}(G)) \qquad (4)$$

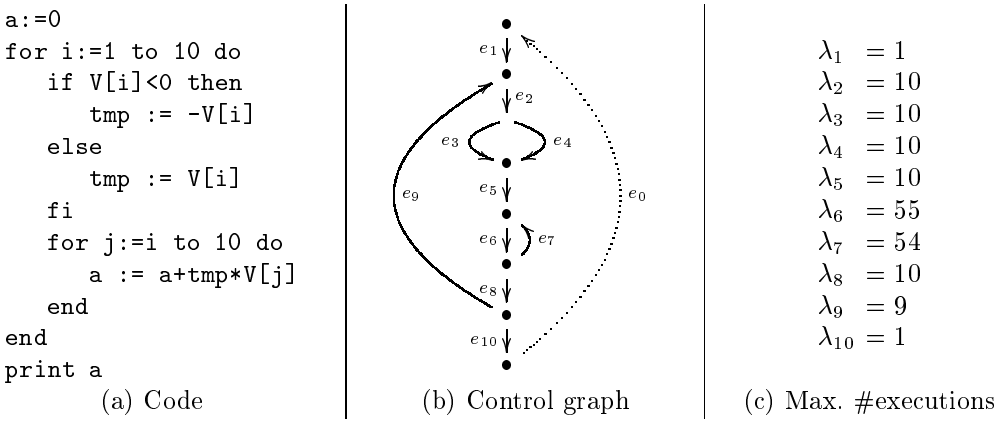
where $\text{in}(n)$ and $\text{out}(n)$ represents the set of incoming and outgoing edges of node n.

For *natural loops* [ASU86, sec. 10.4] (i.e. loops with a single loop entry node n and no jumps into the loop body) a loop entry condition expresses that a loop cannot be executed if none of its preceding edges are executed. In general,

$$\sum_{i \in \text{in}(n) - \text{back}(n)} x_i * \infty \geq \sum_{b \in \text{back}(n)} x_b \qquad (5)$$

[1] The edge e_0 is a purely conceptual construct introduced for the convenience of presentation of the equations below. We define $c_0 = 0$.

[2] We define $x_0 = 1$ to represent that the program is run once.



(a) Code (b) Control graph (c) Max. #executions

Figure 1: An example program

where back(n)$\subset$in(n) represents the incoming edges to n which are backward edges.

In addition to the (mandatory) constraints derived from the structure of the code (equations 3, 4 and 5), any number of (optional) constraints reflecting code dependencies can be added. These additional constraints can be the result of source code analysis or annotations. For instance, consider two consecutive if-statements where the then-statements are mutually exclusive and the two edges e_1 and e_2 correspond to the respective then-statements. The mutual exclusion of these two edges is expressed by the constraint $x_1 + x_2 \leq 1$.

To find the WCET of the program we need to find the maximum value of equation 1 under the constraints we defined in equations 3, 4 and 5, plus any optional constraints. This composes a set of linear equations.

The presented scheme is based on the assumption that the execution time for each basic block can be determined in isolation. There are no means to consider timing variations resulting from pipelined execution and cache memory, since these costs represent inter-block dependencies. In the next section, we describe an extension to this scheme which allows us to take such variations into account.

2.2 Representing the Execution History

To be able to model the effect of architectural components such as pipelines and caches, we need to know the state of those components when we enter a basic block. If the state is known we are able to estimate the gains made from using the component. In sections 3 and 4 we will describe how to determine the state of the pipeline and the cache respectively, and how the execution time for a basic block can be decreased accordingly. In this section we show how history dependent execution times are taken into account in the definition of the execution time ET^P (equation 1).

If we know that the predecessor of edge e_j is e_i (in the execution P) and we know the state of the cache and/or the pipeline after edge e_i then we can calculate the execution time for edge e_j using that state information or, equivalently, we can calculate the *gain* of executing edge e_i before edge e_j, denoted by $g^P_{i\to j}$. Using the cache/pipeline modeling, it is also possible to calculate their states after the execution of edge e_j, allowing calculation of the gain $g^P_{j\to k}$ for successors e_k to e_j.

In an execution P, consider a node n. Let c_i be an incoming edge to n, and c_j an outgoing edge. Then, we define $x_{i\to j}$ to be the number of times edge e_i is executed before e_j.

The execution time for edge t_j can now be formulated as

$$t_j = x_j * c_j - \sum_{i\in \text{in}(n)} x_{i\to j} * g_{i\to j} \qquad (\forall j \in \text{out}(n)) \tag{6}$$

Equation 6 is similar to the definition used in equation 1, with the difference that here we remove the gained time from the total execution time. (Naturally we require, as for x_i, that $x_{i\to j} \in \mathbb{N}_0$.)

The $x_{i\to j}$s can in turn be related to the x_is as follows

$$\sum_{i\in \text{in}(n)} x_{i\to j} = x_j \qquad (\forall j \in \text{out}(n)) \tag{7}$$

$$\sum_{j\in \text{out}(n)} x_{i\to j} = x_i \qquad (\forall i \in \text{in}(n)) \tag{8}$$

The meaning of equations 7 and 8 is perhaps best understood by an example. Figure 2(a) shows a node with multiple incoming and multiple outgoing edges. The summation of the $x_{i\to j}$s is depicted in figure 2(b). Consider the leftmost row of the equations in figure 2(b). It is clear that x_1 must be equal to the sum of the executions passing through e_1, i.e. $x_1 = x_{1\to 3} + x_{1\to 4}$. Similarly for the other column and rows.

Note that if the $g_{i\to j}$s are known a priori we still have a system of linear constraints and thus the WCET

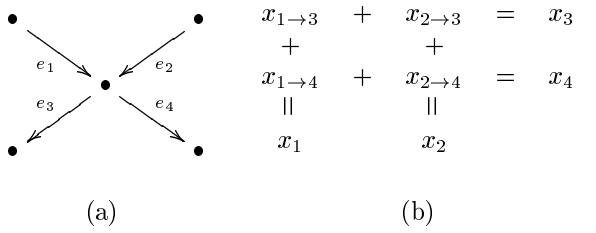


Figure 2: A node with multiple inputs and outputs

(equation 2) can be determined with linear programming.

However, if the $g_{i\to j}$s in turn are dependent on the x_is (or the $x_{i\to j}$s) the equation system is not necessarily linear and other methods for optimization must be used to solve equation 2, see section 5.

3 Pipeline Modeling

In processors with pipelines, instructions are split into multiple execution stages and each stage is executed in sequence. However, different stages from two or more instructions are often executed simultaneously. This *pipeline-overlap* is often quite easily determined for a basic block, using for instance reservation tables [Kog81]. Figure 3 shows an example of a reservation table for a three instruction basic block, executing on a (fictitious) four stage pipelined processor. The vertical dimension represents the different stages in the pipeline, and the time (in clock cycles) evolves from left to right. We assume that the pipeline-overlap inside each basic block is considered when determining the execution time for each basic block (c_i).

Pipeline Stage	Time in clock cycles ⟶ 0	1	2	3	4	5	6	7	8
Instr. fetch	1	1	2	2	3	3			
ALU			1		2	2	2	3	
FPU				1	1	1			
Mem. store							1	2	3

Figure 3: Reservation table for a basic block

For each pair of consecutive basic blocks it is also possible to calculate the pipeline-overlap, using the same reservation tables. However, it is sufficient to consider only the first and last few columns of each basic block [LBJ+95]. (The number of columns which need to be considered is not greater than the maximum number of cycles a single instruction can reside in the pipeline.)

Figure 4 shows how the reservation tables are used to determine the overlap between two blocks i and j. In the reservation tables only the last (or first, respectively) use of a stage is marked. Intuitively, the pipeline-overlap is the distance we can "push" the two reservation tables together without the instructions interfering with each other. The formulas in figure 4 show how the overlap is calculated (where n is the number of columns in basic block i); In this example the overlap is 1 clock cycle.

Let $o_{i\to j}$ denote the pipeline-overlap of basic blocks i and j. Then $g^P_{i\to j} = o_{i\to j}$ (for each execution P). The calculation of the pipeline-overlap can be done a priori and thus the $g^P_{i\to j}$ are all constants and equation 2 is still composed of a set of linear equations.

In this section we have assumed that the pipeline-overlap never reduces the execution time of a basic block below zero. However, it is possible (for instance in a deeply pipelined processor) that $g_{i\to j} > c_j$ if the basic block j does not use all the stages of the pipeline. There are two ways of resolving such situations.

- Prevent the situation from arising. This is done by ensuring that each basic block allocates each pipeline stage at least once, i.e. insert a dummy '×' in the reservation table.

- Use a more sophisticated model of the state of the pipeline when exiting a basic block. For instance, a method similar to the cache modeling in section 4 could be employed.

4 Cache Modeling

A cache complicates the task of estimating the execution time for a sequence of instructions, due to the fact the execution time of a single instruction is dependent on the current content (state) of the cache. To make a safe estimation of WCET we have to treat each memory access as a cache miss unless we can be sure that the referenced block resides in the cache. When calculating c_i there is no knowledge of the state of the cache, and each memory access must be treated as a cache miss.

To remedy this, we will model the cache and the gains from using the cache. As in the case with pipelined execution, our framework calls for an estimation of $g_{i\to j}$, i.e., the gain of using the cache in edge e_j, provided that edge e_i is the immediately preceding edge. Thus,

$$g_{i\to j} = h_{i\to j} * CMP$$

where $h_{i\to j}$ is the number of cache hits when executing edge j coming from edge i, and CMP is the *Cache Miss Penalty*, defined as

$$CMP = \text{memory access time} - \text{cache access time}$$

The following two sections will describe how to model references to the cache in order to determine the number of hits and how to "propagate" cache states between basic blocks in the control graph.

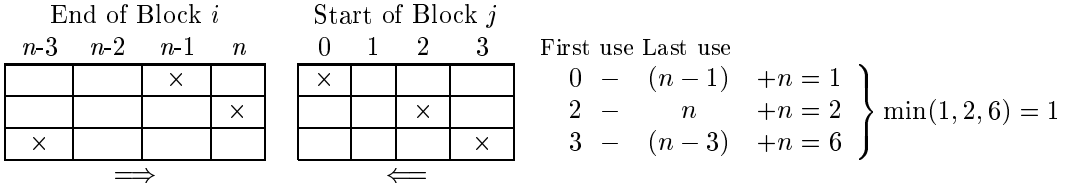


Figure 4: Pipeline-overlap between two basic blocks

The modeling technique presented in the following section is suitable for systems where the *CMP* is equivalent for read and write accesses. (In the general case with different *CMP* for read and write, each memory reference has to be marked with its type and both types have to be handled differently.)

4.1 Memory References and Cache Hits

In this section we describe how a memory cache is modeled. As an example we consider a *2-way set-associative* cache with *Least Recently Used* (LRU) discard policy. A two-way set-associative cache maps each (fixed size) memory block to one cache set. Each set is capable of storing two memory blocks. The LRU discard policy means that the memory block least recently used in a cache set is the block discarded when a new block is loaded to the set.

The technique we use here is applicable to n-way set-associative caches, including $n = 1$ (i.e. direct mapped caches) and n equal to the size of the cache (i.e. fully associative caches).

We annotate each edge e_i in the graph G with a list of memory references $R_i = r_1, \ldots, r_m$ made in that basic block. We also assign to the end of each edge e_i a cache C_i. Then we calculate the cache hits $h_{i\to j}$ for each pair of successive edges by simulating the references of edge e_j in the cache C_i.

First, consider a single reference to a memory block r, which is mapped to cache set s in cache C. The cache C contains the sets $C(1) \ldots C(p)$. Each set stores two memory blocks, such that $C(i) = [b_1, b_2]$ where memory block b_2 is the least recently used. Each resulting cache set $C'(i)$ is described by

$$\begin{aligned} i = s \Rightarrow & C'(i) = \begin{cases} [r, b_2] & \text{if } r = b_1 \\ [r, b_1] & \text{otherwise} \end{cases} \\ i \neq s \Rightarrow & C'(i) = C(i) \end{aligned} \tag{9}$$

i.e. all cache sets are copied from C to C' except s which is updated with the contents of memory block r.

Let $h(C, r)$ indicate whether a reference r to a cache C is a hit or miss. Formally,

$$h(C, r) = \begin{cases} 1 & \text{if } r \in C(s) \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Multiple references $R_j = [r_1, \ldots, r_m]$ are modeled in sequence, $C^0_{i\to j} \overset{r_1}{\curvearrowright} C^1_{i\to j} \overset{r_2}{\curvearrowright} \ldots \overset{r_m}{\curvearrowright} C^m_{i\to j}$ (where $C^0_{i\to j} = C_i$), and the number of cache hits in such a sequence is the sum of the hits, i.e.

$$h_{i\to j} = \sum_{0 \le k \le m-1} h(C^k_{i\to j}, r_k) \tag{11}$$

Note that equations 9 and 10 are purely declarative statements and knowledge about the actual values of r_k is not required. When r_k is an instruction reference its value can be statically determined but when r_k is a data reference its value may be unknown or partially known.

Unknown or partially known references are not a problem when using this technique. However, the tightness of the estimated WCET is dependent on the information known about the references. Partial information about references can be ranges or sets of possible values for the reference or dependencies between different references. For instance, if references r_1 and r_2 are unknown but it is known that they reference the same memory block, i.e. $r_1 = r_2$, then the reference simulation will report one cache miss (for the unknown reference r_1) followed by one cache hit (for r_2, which is known to be in the cache).

4.2 Cache Propagation

Besides modeling the cache references, we also need to define the state of the *propagated cache* C_j, i.e. the state of the cache assigned to the end of edge e_j. This is done by selecting as C_j one of the updated caches from an incoming edge with positive flow x_i. Formally

$$C_j \in \{C^m_{i\to j} | i \in \text{in}(n) \wedge x_i \neq 0\} \tag{12}$$

If there is no incoming flow, that is, all x_i are zero, then x_j is also 0 and C_j is undefined.

The maximization in equation 2 will force C_j to take the value which causes the largest total execution time. For the first edge of a loop e_j, the worst value of the C_j is probably one of the $C^m_{i\to j}$ corresponding to an edge e_i that is not a backward edge in the loop. This will cause the successors of e_j in the loop to use a cold cache each iteration, which leads to pessimistic WCET estimation. To enable the subsequent iterations in the loop to benefit from the memory blocks loaded into the

cache during the first iteration it is necessary to *unroll* the loop one round when producing the graph G.

4.3 Optimizing over Cache Models

In contrast to the pipeline-overlap calculation illustrated in section 3, the calculation of the number of cache hits and the states of the caches cannot be done for each basic block a priori. The reason for this is that the state of a cache depends not on the current basic block but rather on the execution history.

The formulas 9, 10, 11 and 12 expressing cache references, cache hits and cache propagation will be part of the expression for ET. This calls for other maximization methods than linear programming. For instance, constraint satisfaction over finite domains (see section 5) can now be used to optimize equation 2.

Section 6 will show, for an example program, the results obtained for the cache modeling.

5 Constraint Satisfaction

We will here give a short and very introductory guide to the technique called *constraint satisfaction*. For a more complete description, see [Tsa93, Kum92].

A large variety of problems can be viewed as constraint satisfaction problems (CSP), e.g. scheduling, planning and allocation. A CSP is formulated as a set of variables and a set of constraints; solving a CSP amounts to finding values for the variables such that all constraints are satisfied. A number of different approaches for solving these problems have been developed, most of them being a combination of *backtracking search* and *constraint propagation*.

Search involves the systematic and exhaustive trying of values for variables. Constraint propagation involves removing infeasible values from variable domains, making active use of the constraints. As these two techniques are combined, the constraint propagation prunes the search tree in an a priori way.

Support for constraint satisfaction has been added to most major programming languages, including imperative and object-oriented, functional and logic-based.

Constraint Satisfaction is related to Linear Programming in the sense that they sometimes attack the same problems, but the techniques as such are quite different. Linear Programming is based on a deterministic manipulation of equations, while constraint satisfaction is based on constrained search. Despite the fact that the complexity of constrained search is exponential in worst case, many useful problems have proved to be solvable with good performance.

6 Implementation and Evaluation

The technique presented in sections 2 through 4 has been implemented using the finite domain extension of SICStus Prolog [C+95]. All variables of the equations representing the control graph from section 2 have been represented as finite domain variables, along with all variables modeling the cache from section 4 (we ignore pipelines for this example). The equations themselves have been expressed as finite domain constraints, using the arithmetical and logical formulas from section 2 and 4. Using the constraint solving engine in SICStus, we optimize for the maximum value of the expression ET^P (equation 2).

In this example we emphasize the modeling of the cache, and we therefore define $c_i = 0$ and $CMP = 1$. Thus, worst case execution path will be the path with fewest cache hits.

Consider the example program from figure 1. In this experiment we need, besides the control graph, the memory references made in each edge. These are obtained from the (pseudo assembler) instructions in figure 5(a) using the memory allocation shown in figure 5(b). The assembler code in figure 5(a) is generated by straightforward, and naive, manual compilation of the code in figure 1(a).

The address of all memory references are known (and shown in figure 5(a)) except the array references `V[i]` and `V[j]` (r_1, r_2, r_3 and r_4) which are unknown due to the index which varies. But, we do know that for all executions and every traversal of edge e_2 that `V[i]` is the same reference as `V[i]` in edges e_3 and e_4 (i.e. $r_1 = r_2 = r_3$). This information can, using our method, be taken into account, and our results show that the information leads to more predicted cache hits (i.e. a closer estimation).

Figure 6 illustrates the number of cache hits in the worst case execution in three different situations. First we have column *Naive* which is used as a lower reference mark, where we have not used the cache propagation from section 4.2. Instead, cold caches are used for basic blocks with multiple incoming edges, and only in the trivial case when a basic block has a single predecessor are caches propagated. The second column, *Unknown*, shows the effects with fully propagating caches (i.e. equation 12 is used for selecting a cache to propagate), and finally, column *Same* shows the result when the knowledge that $r_1 = r_2 = r_3$ is utilized.

The cache used is a cache with 8 cache sets, and the cache is used for both instruction and data references, i.e. , it is a unified 2-way set-associative cache. For simplicity we use the same size for memory blocks, machine instructions and integers.

The accumulated cache hits shown in the last row of figure 6 show clearly that cache propagation between all basic blocks is essential to get a good cache hit es-

	Pseudo Assembler	Ref. Address
e_1	store$_1$ `a` store$_2$ i	1 45 2 46
e_2	load$_3$ `i` load$_4$ `V[i]` cmp$_5$ goto$_6$	3 46 4 r_1 5 6
e_3	load$_7$ `i` load$_8$ `V[i]` neg$_9$ store$_{10}$ `tmp` goto$_{11}$	7 46 8 r_2 9 10 48 11
e_4	load$_{12}$ `i` load$_{13}$ `V[i]` store$_{14}$ `tmp`	12 46 13 r_3 14 48
e_5	load$_{15}$ `i` store$_{16}$ `j`	15 46 16 47
e_6	load$_{17}$ `j` load$_{18}$ `V[j]` load$_{19}$ `tmp` mult$_{20}$ load$_{21}$ `a` add$_{22}$ store$_{23}$ `a` load$_{24}$ `j` add$_{25}$ store$_{26}$ `j` cmp$_{27}$	17 47 18 r_4 19 48 20 21 45 22 23 45 24 47 25 26 47 27
e_7	goto$_{28}$	28
e_8	load$_{29}$ `i` add$_{30}$ store$_{31}$ `i` cmp$_{32}$	29 46 30 31 46 32
e_9	goto$_{33}$	33
e_{10}	load$_{34}$ `a` print$_{35}$	34 45 35

(a) Memory references

Code	1
⋮	⋮
	35
`a`	45
`i`	46
`j`	47
`tmp`	48
`V[1]`	49
⋮	⋮
`V[10]`	59

(b) Memory allocation

Figure 5: Memory references for the example program

Flow	*Naive*		*Unknown*		*Same*	
$i \to j$	$x_{i\to j}$	$h_{i\to j}$	$x_{i\to j}$	$h_{i\to j}$	$x_{i\to j}$	$h_{i\to j}$
$0 \to 1$	1	0	1	0	1	0
$1 \to 2$	1	1	1	1	1	1
$2 \to 3$	11	0	0	0	11	0
$2 \to 4$	0	0	11	0	0	0
$3 \to 5$	11	0	0	1	11	0
$4 \to 5$	0	0	11	0	0	0
$5 \to 6$	11	2	11	4	11	5
$6 \to 7$	55	0	55	0	55	0
$6 \to 8$	11	1	11	2	11	2
$7 \to 6$	55	2	55	12	55	12
$8 \to 9$	10	0	10	0	10	0
$8 \to 10$	1	0	1	1	1	1
$9 \to 2$	10	1	10	2	10	2
	Hits: 154		Hits: 748		Hits: 759	

Figure 6: Flow and cache hits in worst case execution

timation. Propagating within basic blocks and between consecutive basic blocks only, as in *Naive*, is not sufficient.

We can also conclude from comparing the last two columns that a closer cache hit estimation can be achieved by utilizing available information about references, even if they are partially unknown.

Unknown or partially unknown references are computationally expensive. Calculating WCET for this example is done in a few seconds if all references are known. Having partially known references like in *Same*, the calculation takes a couple of minutes while fully unknown references (as in *Unknown*) extends the analysis time to several hours. This is partly due to the fact that only little work has been spent improving the performance, but also due to the complexity of the problem as such. Unknown references cannot be mapped to a specific cache set. This severely inhibits the propagation of cache states during optimization, since no cache set in an edge with an unknown reference can be determined without search.

7 Conclusion and Future Work

We have presented a method which allows the behavior of modern hardware architecture to be considered when determining the Worst Case Execution Time (WCET) of a program. Our method is based on the implicit path enumeration technique (IPET) [LM95, PS95], which we have extended so that *history dependent execution times* can be considered. This allows us to calculate close WCET estimations on cached and pipelined architectures.

We have shown how a unified 2-way set associative cache can be modeled with our technique, extending the now well understood problem of modeling direct mapped instruction caches. Our technique allows us to model n-way set-associative caches and references to memory where the exact address is unknown, or partially known. This makes a unified analysis of instruction and data cache references possible.

Although the presented method uses detailed knowledge about the history of execution, tighter WCET estimations could probably be made if even more information of the execution path were explicitly represented. By defining $x_{i\to j}$ for not only successive edges but for all pair of edges where there exists a path through both edges e_i and e_j we could avoid the need for unrolling of the first loop round to get a tight WCET estimation. Also, by explicitly representing how the cache state changes during execution of a loop it should be possible to obtain even better precision for the data cache modeling.

Our technique is most closely related to the recent

work of Li, Malik and Wolfe [LMW96]. Their approach is based on breaking down the cache modeling to a set of cache conflict graphs (one for each cache set), transforming the problem to a linear network flow problem which can be solved using Integer Linear Programming (ILP) [NRT89]. The experiments with instruction caching in [LMW96] show promising results in terms of accuracy of the WCET prediction and in most cases the analysis time is negligible. Though for some of their test-cases the analysis time is several hours. Their method is applicable for data caching, however, no performance results for data caching is reported. In [LMW96] data caching of references with unknown (or partially known) addresses result in cache conflict graphs with a high degree of connectivity and loose linear constraints, which might increase analysis time and introduce performance problems.

In contrast to linear constraints and cache conflict graphs, we model the cache (as well as other micro-architectural properties) straight-forwardly making use of the non-linear arithmetic and logic formulas made available by the constraint satisfaction approach of a finite domain constraint solver [C+95]. When only considering references to known memory addresses (e.g. instruction caching) we experience (for our limited test-cases) good performance, both in terms of WCET tightness and analysis time. Also, when knowledge of the actual addresses of the references is limited (which is often the case for data caches) our method make use of the available information to tighten the WCET. However, limited information severely affects the analysis time of the tool, and when addresses are totally unknown the execution time might be in the order of hours even for small programs.

Our presented technique is not yet mature enough to be used in practice. It suffers from performance problems when attacking large programs or loosely constrained references (such as many data references where the actual memory address is unknown). Its virtue, on the other hand, is that it is simple and has powerful expressiveness. It handles instruction and data cache references uniformly, and it is straight-forward to extend to handle other micro-architectural entities than pipeline and cache. While we haven't done a real effort in enhancing the performance, such a task is the obvious choice for future work aiming at validating the usefulness of this technique.

References

[AMWH94] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proc. 15th Real-Time Systems Symposium*, pages 172–181, December 1994.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986. ISBN 0-201-10194-7.

[BMSO+96] J.V. Busquets-Mataix, J.J. Serrano, R. Ors, P.Gil, and A. Wellings. Adding instruction cache effects to schedulability analysis of preemptive real-time systems. In *Proc. IEEE Real-Time Technology and Applications Symposium*, pages 204–212. IEEE Computer Society Press, June 1996.

[BN94] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *Proc. of the 1st ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[C+95] Mats Carlsson et al. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, June 1995. Release 3.

[CBW94] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of SPARK ADA. In *Proc. ACM Sigplan Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[Gus94] J. Gustafsson. Calculating of execution times in object-oriented real-time software - a study focused on RealTimeTalk. Damek, KTH Stockholm, February 1994. Lic. thesis.

[KMH96] S-K. Kim, S.L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proc. IEEE Real-Time Technology and Applications Symposium*, pages 230–240. IEEE Computer Society Press, June 1996.

[Kog81] Peter M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, New York, 1981.

[Kum92] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *A.I. Magazine*, 13(1):32–44, Spring 1992.

[LBJ+95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[LHS+96] C. Lee, J. Han, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proc. 17th Real-Time Systems Symposium*, 1996.

[LL94] J-C. Liu and H-J. Lee. Deterministic upperbounds of the worst-case execution time of cached programs. In *Proc. 15th Real-Time Systems Symposium*, pages 182–191, December 1994.

[LM95] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM Workshop on Lang., Comp. and Tools for RTS*, May 1995.

[LMW96] Y-T S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th Real-Time Systems Symposium*, pages 254–263. IEEE, IEEE Computer Society Press, December 1996.

[NRT89] G. L. Nemhasuer, A. H. G. Rinnooy Kan, and M. J. Todd. *Optimization*. North-Holland, 1989.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, pages 159–176, 1989.

[PS91] C.Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *IEEE Computer*, May 1991.

[PS95] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Techn. Univ., Inst. für Technische Informatik, Vienna, April 1995.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.