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

ABSTRACT

We present a method for determining the worst case execution time (WCET) for programs running on systems with modern hardware architectures, e.g. pipelined processors and cache memory. The method is based on the implicit path enumeration technique (IPET), in which all possible execution paths of a program are described by placing constraints on, and relations between, the number of executions of different parts of the program. The execution times for all paths are expressed in terms of those numbers and the maximum execution time is found using optimization (e.g. linear programming).

In this paper we present an extension of the IPET which allows different micro-architectural aspects of the execution of programs to be modeled independently. The models are later unified when estimating the WCET. As examples, we show how to model pipelined execution and set-associative caches.

Keywords: Hard real-time systems, Worst case execution time, Cache memory, Pipelined execution, Constraint satisfaction.

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.

If the resource requirements can be closely estimated it is possible to use cheaper hardware platforms for these systems. Many modern real-time systems are mass produced, e.g. embedded software in automobiles and mobile phones. Hence, it is possible to cut production costs for a real-time system by making closer estimates of its resource requirements.

In this paper we study the problem of statically (off-line) determine an upper bound on the execution time of a program running on a single computer with moden hardware architecture. We present a mothod which allow a variety of micro-architectural components to be modeled. Specifically, we present methods to model pipelined processors and cache memory.

In general, a real-time system often consists of multiple programs with execution split over multiple computers. The global analysis of those systems are often referred to as *scheduling analysis*: a system is said to be schedulable if it can be shown that all timing requirements will be met. Examples of such analysis are presented in [LL73, ABT+93, SSNB95]. All these analysis methods require that the WCET of each task (i.e. program) be known. The traditional methods for scheduling analysis do not handle systems with cache memory very well. The reason for this is that they do not capture the inter-task cache interference, i.e. they do not account for the delay caused when a preempted task gets some (or all) of its cache contents replaced by the preempting task. To make a safe analysis the solution so far has been to assume that each memory reference is a cache miss. This (pessimistic) assumption simplifies both the schedulability analysis and the WCET estimation, at the cost of a pessimistic resource requirements estimation.

However, recently several approaches has been made to bound this inter-task interference [BN94, BMSO+96, LHS+96]. To be able to utilize those enhanced schedulability analysis methods (and minimize the overall resource requirements) we need to estimate the intra-task cache interference and account for the decreased execution time of a program when cache memory is used.

When estimating the execution time for a hard real-time program it is important that the estimate is *safe* (i.e. not below the actual WCET) and *tight* (i.e. a close approximation of the actual WCET). Analysis techniques which do not consider the cache, e.g. [PK89, PS91, Gus94, CBW94, PK95, LM95], are not able to provide tight estimations of WCET for cached systems.

Recently, researchers have proposed methods which allow tighter estimation of WCET in systems which use caches, e.g. [AMWH94, LL94, LBJ+95, KMH96, LMW96]. Arnold et.al. [AMWH94] consider only instruction caches which partly limits 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 express 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] (on which our work too is based) 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] all the possible execution paths of a program is described by placing constraints on, and relations between, the number of executions different parts of the program. The execution time of each execution is expressed in terms of those numbers and the maximum execution time is found by maximizing the expression for the execution time using (for instance) linear programming (see section 2).

IPET allows semantical dependencies to be expressed as constraints on the program's control flow graph. 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 approach to estimate the WCET has many similarities to the method presented in [LMW96] and essentially we tackle the same problems (i.e. pipeline and cache modeling). However, 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 speeds up consecutive memory addresses), within the same framework.

Paper Outline: Section 2 describes the basic method we use to calculate the WCET; section 2.1 recapitulates the foundations of this method as published before while section 2.2 describes our extensions to the method. Sections 3 and 4 describe how the pipeline and cache effects are estimated and incorporated in the method from section 2. In section 5 we give an example of how cache effects affects the WCET estimation. We conclude our work in section 6.

2 Calculating the Worst Case Execution Time

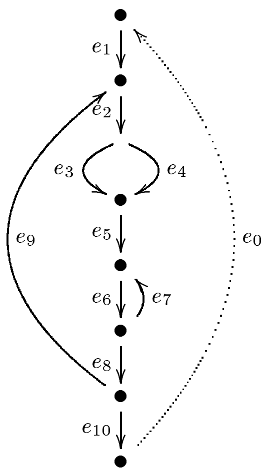
In this section we describe a method for determining the Worst Case Execution Time (WCET) of a program. The method is based on application of linear programming methods on a WCET-expression derived from the control-graph of the program [LM95, PK95]. We begin our description by recapitulating the work described in [PK95] (section 2.1). We then describe our extension to that method (section 2.2), which allows us to consider cache and pipeline effects in the execution time analysis.

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

```
a:=0
for i:=1 to 10 do
   if V[i]<0 then
      tmp := -V[i]
   else
      tmp := V[i]
   fi
   for j:=i to 10 do
      a := a+tmp*V[j]
   end
end
print a
```

(a) Code



(b) Control Graph

$$\begin{aligned}
\lambda_1 &= 1\\
\lambda_2 &= 10\\
\lambda_3 &= 10\\
\lambda_4 &= 10\\
\lambda_5 &= 10\\
\lambda_6 &= 55\\
\lambda_7 &= 54\\
\lambda_8 &= 10\\
\lambda_9 &= 9\\
\lambda_{10} &= 1
\end{aligned}$$

(c) Max. #executions

Figure 1: An example 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 by semantic analysis of the program, 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 actual 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 execution time of the program (ET^P) is the sum of the times spent in each edge on path P, i.e.

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

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

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

For notational convenience we ignore the superscript ^{P}, when the meaning is clear, for the rest of this paper.

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

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

$$x_i \in \mathbb{N}_0, \text{ and } x_i \leq \lambda_i \tag{3}$$

However, these restrictions on the x_is are not enough to characterize the possible executions of the program. In [LM95, PK95], 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, thus the maximization in equation 2 can be done with linear programming (see e.g. [NRT89]).

The most fundamental constraints express the preservation of flow. This 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]

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

where in(n) and out(n) represents the set of incoming and outgoing edges of node n.

However, since a program may contain loops the above constraint is not enough to characterize precisely the valid executions of a program. We need to represent (explicitly) the fact that a loop can only be executed if a basic block immediately preceding the loop entry is executed. In figure 1 the node succeeding e_1 represents a loop entry point and the loop entry condition can be expressed by $x_1 * \lambda_9 \geq x_9$. This way, if x_1 is zero (i.e. no incoming flow) then x_9 (and all other edges inside the loop) must also be zero.

Here we consider *natural loops* [ASU86, sec. 10.4], i.e. loops with a single loop entry node[3] n and one or more backward edge e_b. Node n also has one or more incoming edges, e_i (distinct from the e_bs) and one or more outgoing edges e_j. Figure 2 shows a sample loop entry node.

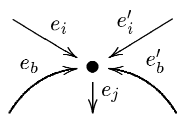


Figure 2: A sample loop entry node for a natural loop

The general form of the loop entry entry condition is

$$\sum_{i \in \mathrm{in}(n)-\mathrm{back}(n)} x_i * \sum_{b \in \mathrm{back}(n)} \lambda_b \geq \sum_{b \in \mathrm{back}(n)} x_b \qquad (\forall n \in \mathrm{nodes}(G)) \tag{5}$$

[2]We define $x_0 = 1$ to represent that the program is run once.
[3]This restriction essentially prohibits jumps *into* a loop.

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

In addition to the (mandatory) constraints stated above, 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 corresponds to the respective then-statements. The mutually 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 can be done with linear programming [LM95, PK95].

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 interblock 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 History of Execution

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 should be able to estimate the gains made from using the pipeline or the cache. 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).

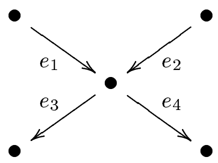
If we know that the predecessor of edge e_j is e_i (in the execution P) and we know the state of the cache and/or the pipeline in edge e_i then (assuming we can model the cache/pipeline) we can calculate an execution time for edge e_j using that state information or, equivalently, we can calculate the *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 the gain $g^P_{j \to k}$ (for successors e_k to e_j) to be calculated.

Consider each possible execution path through a node n in an execution P. Let $x^P_{i \to j}$ denote the number of times edge e_i is executed before e_j (where $e_i \in \text{in}(n)$ is an incoming edge and $e_j \in \text{out}(n)$ is an outgoing edge of node n).

Again, ingoring the ^{P} superscript, we define the formulae for t_j 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 we



$$\begin{array}{ccccc}
x_{1\to 3} & + & x_{2\to 3} & = & x_3 \\
+ & & + & & \\
x_{1\to 4} & + & x_{2\to 4} & = & x_4 \\
\| & & \| & & \\
x_1 & & x_2 & &
\end{array}$$

(a) (b)

Figure 3: A node with multiple inputs and outputs

here remove the time gained 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: Consider one outgoing edge j, then it's clear that the sum of all executions leading to edge j must be equal to x_j. Formally

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

Similarly, the number of executions of an edge $i \in \mathrm{in}(n)$ must be equal to the sum of all executions passing through i. Formally

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

The intuition behind equations 7 and 8 is perhaps best understood by an example. Figure 3(a) shows a node with multiple incoming and multiple outgoing edges. The summation of the $x_{i\to j}$s is depicted in figure 3(b). Consider the leftmost row of the equations in figure 3(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 (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 *constraint satisfaction methods* must be used to solve equation 2.

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 4 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 $\longrightarrow$ 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 4: Reservation table for a basic block with 3 instructions

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 5 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 5 shows 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.

End of Block i

n-3	n-2	n-1	n
		$\times$	
			$\times$
$\times$			

$\Longrightarrow$

Start of Block j

0	1	2	3
$\times$			
		$\times$	
			$\times$

$\Longleftarrow$

$$\left.\begin{array}{lcl} 0- & (n-1) & +n = 1 \\ 2- & n & +n = 2 \\ 3- & (n-3) & +n = 6 \end{array}\right\} \min(1,2,6) = 1$$

Figure 5: Pipeline-overlap between two basic blocks

Let $o_{i \to j}$ denote the pipeline-overlap of basic blocks i and j. Then $g_{i \to j} = o_{i \to j}$. The calculation of the pipeline-overlap can be done a priori and thus the $g_{i \to j}$ are all constants and it is still possible to use linear programming to find the WCET (i.e. to solve equation 2).

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 do not use all the stages of the pipeline. There are two ways to resolve 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 '$\times$' 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 deployed.

4 Cache Modeling

In processors using memory caches, increased execution speed is acquired by reducing the time for memory accesses. The cache is a special, smaller memory with faster access time which stores memory blocks that are frequently used. The size of the cache is limited, thus only a subset of the memory blocks used can be stored. When a memory block is referenced, it is first checked for in the cache. If the block is there it is a *cache hit*, otherwise it is a *cache miss*. In the case of a miss, the block is loaded to the cache from main memory, replacing some other block. The additional execution time resulting from a cache miss is called the *Cache Miss Penalty*, CMP. The CMP is defined as

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

The 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 the time for each memory access must in this stage include the CMP.

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.

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 edges in the control graph.

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 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. The resulting cache C' is described by

$$\underbrace{C'(s) = \begin{cases} [r, b_2] & \text{if } r = b_1 \\ [r, b_1] & \text{otherwise} \end{cases}}_{r \text{ is the most recently used block}} \wedge \underbrace{\forall i : i \neq s \to C'(i) = C(i)}_{\text{All other blocks are unaffected}} \tag{9}$$

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,

$$h(C, r) = \begin{cases} 1 & \text{if } C(s) = [r, b] \vee C(s) = [b, r] \text{ for any block } b \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

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.

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

Note that equations 9 and 10 are purely declarative statements and knowledge about the actual values of the r_is is not required. When r_i is an instruction reference its value can be statically determined but when r_i 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 $R_i = [r_1, r_2]$ and 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 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

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

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. 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 [FHK+92, Tsa93] can be used to find the WCET (solve equation 2).

The next section will show, for an example program, the results obtained for the cache modeling.

5 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, we optimize for the maximum value of the expression ET (equation 2). Each term of the expression ET is dependent on the gain from using the cache ($g_{i \to j}$ for each flow in every node), which in turn is dependent on the cache hits obtained. So, the search together with the constraint propagation produces the maximum cost by implicitly enumerating paths, and thus also implicitly instantiating and propagating the cache states of those paths.

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 6(a) using the memory allocation shown in figure 6(b). The assembler code in figure 6(a) is generated by straightforward, and naive, manual compilation of the code in figure 1(a).

	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`	7 46 8 r_2 9 10 48
	goto$_{11}$	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$_{20}$	17 47 18 r_4 19 48
	load$_{21}$ `a` add$_{22}$ store$_{23}$ `a` load$_{24}$ `j` add$_{25}$	20 21 45 22 23 45
	store$_{26}$ `j` cmp$_{27}$	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 Alloc.

Figure 6: Memory references for the example program

We assume that all data is allocated statically. Thus, the address of all memory references are known (and shown in figure 6(a)) except the array references `V[i]` and `V[j]` (r_1, r_2, r_3 and r_4 in figure 1(a)) 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. $r_1 = r_2 = r_3$). To illustrate how this fact can be used in the cache analysis, we calculate the number of cache hits in the worst-case execution, both with (column *Same* in figure 7) and without (column *Distinct* in figure 7) the knowledge that $r_1 = r_2 = r_3$. The cache used is a cache with 11 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.

Flow	*Distinct*		*Same*	
$i \to j$	$x_{i \to j}$	$h_{i \to j}$	$x_{i \to j}$	$h_{i \to j}$
$0 \to 1$	1	0	1	0
$1 \to 2$	1	1	1	1
$2 \to 3$	0	2	11	2
$2 \to 4$	11	1	0	2
$3 \to 5$	0	1	11	1
$4 \to 5$	11	1	0	1
$5 \to 6$	11	6	11	6
$6 \to 7$	55	0	55	0
$6 \to 8$	11	2	11	2
$7 \to 6$	55	16	55	16
$8 \to 9$	10	0	10	0
$8 \to 10$	1	1	1	1
$9 \to 2$	10	1	10	1
	Hits: 1002		Hits: 1013	

Figure 7: Flow and cache hits in worst-case execution

The number of cache hits obtained are the same for every edge between the runs, except for hits $h_{2 \to 4}$, which increases from 1 to 2 when we know that $r_1 = r_2 = r_3$. In figure 7 we see that this actually changes the worst-case path from going through e_4 to going through e_3 (by studying the values of $x_{i \to j}$).

Computationally, unknown or partially unknown references are very expensive. Calculating WCET for this example, including the caches, 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 *Distinct*) extends the optimization 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 line, and partially known references are mapped to a set of cache lines. This severely inhibits the propagation of cache states during optimization, since no cache line from an edge with an unknown reference can be determined without search.

6 Conclusion and Future Work

The implicit path enumeration approach [LM95, PK95] to worst-case execution time analysis has the benefit of allowing complex semantical dependencies to be modeled. Using this technique as a basis, we describe the control flow more detailed by explicitly representing the history of the execution (with $x_{i \to j}$). Using the execution history we show how architectural aspects such as caches and pipelines can be modeled. Unified, these contributions build a general framework in which semantical and architectural analysis can be joined. Our approach allows us to go beyond the linear constraints used in the original technique; using logical constraints (formulas) to model the micro-architectural properties.

Our cache model extends 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 allow us to unify analysis of instruction and data memory references.

Albeit 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 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 instruction cache experiments in [LMW96] shows 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, this will probably increase analysis time and may 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 caching) our method make use of limited information to tighten the WCET. However, limited information severely affects the analysis time of the tool, when addresses are totally unknown the execution time is in the order of hours (for the example in figure 1).

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 time performance, such a task is the obvious choice for future work aiming at validating the usefulness of this technique.

References

[ABT+93] N.C. Audsley, A. Burns, K.W. Tindell, M.F. Richardson, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[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.

[FHK+92] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint logic programming: An informal introduction. In G. Comyn, N. E. Fuchs, and M. J. Ratcliffe, editors, *Logic Programming in Action*, LNCS 636, pages 3–35. Springer-Verlag, 1992.

[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.

[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. 17^{th} Real-Time Systems Symposium*, 1996.

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LL94] J-C. Liu and H-J. Lee. Deterministic upperbounds of the worst-case execution time of cached programs. In *Proc. 15^{th} 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. 17^{th} 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.

[PK95] P. Puschner and C. Koza. Computing maximum task execution times with linear programming techniques. Technical report, Report, Techn. Univ., Inst. für Technische Informatik, Vienna, May 1995.

[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.

[SSNB95] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, pages 16–25, June 1995.

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