Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software

Yau-Tsun Steven Li
yauli@ee.princeton.edu

Sharad Malik
sharad@ee.princeton.edu

Andrew Wolfe
awolfe@ee.princeton.edu

Department of Electrical Engineering, Princeton University,
Princeton, NJ 08544, USA.

Abstract

Real-time systems are characterized by the presence of timing constraints in which a task must be completed within a specific amount of time. This paper examines the problem of determining the bound on the worst case execution time (WCET) of a given program on a given processor. There are two important issues in solving this problem: (i) ***program path analysis****, which determines what sequence of instructions will be executed in the worst case, and (ii)* ***microarchitecture modeling****, which models the hardware system and determines the WCET of a known sequence of instructions. To obtain a tight estimate on the bound, both these issues must be addressed accurately and efficiently. The latter is becoming difficult to model for modern processors due to the presence of pipelined instruction execution units and cached memory systems.*

Because of the complexity of the problem, all existing methods that we know of focus only on one of above issues. This limits the accuracy of the estimated bound and the size of the program that can be analyzed. We present a more effective solution that addresses both issues and uses an integer linear programming formulation to solve the problem. This solution is implemented in the program `cinderella` [1] *which currently targets the Intel i960KB processor and we present some experimental results of using this tool.*

1 Introduction

The execution time of a program running on a given system may vary significantly according to different input data values and initial system state. In many cases it is essential to know the worst case execution time (WCET) of a program running on a particular hardware system. This WCET is useful in many areas. In hard real-time systems, the designer must prove that the WCET satisfies the timing deadlines. Many real-time operating systems rely on this information for process scheduling. In embedded system designs, the WCET of the software is often required for deciding how hardware/software partitioning is done.

The *actual* WCET of a program cannot be determined unless we simulate all possible combinations of input data values and initial system states. This is clearly impractical due to the huge number of simulations required. As a result, we can only obtain an estimate on the actual WCET by performing a static analysis of the program. For it to be useful, the *estimated* WCET must be tight and conservative such that it bounds the actual WCET without introducing undue pessimism.

The objective of this paper is to examine the problem of determining the estimated WCET of a given program on a given hardware system, assuming uninterrupted execution. There are two components involved in solving this problem. They are:

1. **Program path analysis**. This determines what sequence of instructions will be executed in the worst case scenario. Infeasible program paths should be removed from the solution search space as much as possible. This can be done by a data flow analysis of the program, but is more effective with the help of programmer. Therefore, the analysis should provide a mechanism for program path annotations. Another important aspect is that the number of program paths is typically exponential with the program size. An efficient path analysis method is required to avoid exhaustive program path search.

2. **Microarchitecture modeling**. This models the hardware system and computes the WCET of a given sequence of instructions. This is becoming difficult to model because most modern processors have pipelined instruction execution units and cached memory systems. These features, while speeding up the typical performance of the system, complicate timing analysis. The execution time of a single instruction depends on many factors and varies more than in the previous generation of microprocessors. The cache memory is particularly difficult to model accurately. To determine whether or not the execution of an instruction results in a cache hit, several previously executed instructions must be examined. Any incorrect prediction will result in large pessimism.

Both components must be studied well and together in order to obtain a tight estimated WCET. We will study program path analysis in Section 3 and microarchitecture modeling in Section 4.

[1] In recognition of her hard real-time constraint — she had to be back home at the stroke of midnight!

2 Related Work

The problem of determining a program's estimated WCET is in general undecidable and is equivalent to a halting problem. Kligerman and Stoyenko [1], as well as Puschner and Koza [2] have listed the conditions for this problem to be decidable. These conditions are: (i) absence of recursive function calls, (ii) absence of dynamic structures and (iii) bounded loops. These restrictions can be imposed either through specific language constructs or through an external annotation mechanism. These researchers adopt the first approach. They develop the languages Real-Time Euclid and MARS-C respectively. The main drawback of this approach is that it comes with the usual high costs associated with a new programming language. Mok *et al.* [3], and Park and Shaw [4] choose the latter approach. The loop bounds and other path information are annotated in a separate file. The analyzing tool reads the annotated file and the executable code of the program and then computes the estimated WCET. We believe this is a better approach, as it does not limit the choice of programming language and it requires only minimal additional programming tools.

Most researchers have recognized that in order to tighten the estimated WCET, it is necessary to remove some logically infeasible program paths. While some of them can be automatically inferred from the program using data flow and symbolic analysis techniques, it is widely felt that this is a difficult task. In contrast, it is relatively less difficult for the programmer to provide such information since he/she is familiar with what the program is supposed to do. The scope of the path information that the programmer can provide may have a direct impact on the tightness of the estimated WCET. Initial efforts [2, 3] to include this information were restricted to providing loop bounds and maximum execution counts of program statements. Subsequent work by Park and Shaw accepts path annotations regarding interactions among program statements (e.g., two statements are mutually exclusive). They use regular expressions to represent all feasible program paths. While this approach is powerful in describing program paths and can evaluate the worst case program path, it is computationally expensive. As a result, several compromises are made to limit both the scope of user annotations and the tightness of the analysis.

Microarchitecture modeling handles the timing analysis of a known sequence of instructions. It is widely agreed that this must be done at the assembly level in order to capture all the effects of compiler optimizations and the microarchitecture implementation. Previously, the modeling was simple because the execution time of an instruction was largely independent of others. The researchers normally assumed that the execution time of an instruction was a constant and was equal to its worst case execution time throughout the execution of the program. In modern processors, however, the execution time of an instruction depends on its surrounding instructions and it varies more than in the previous generations of processors. Using the above simple model will result in a very loose estimated WCET. Worse, as the processors become more complicated, the pessimism due to inaccurate modeling is becoming a dominant factor and is increasing. This problem must be overcome before modern processors can be used efficiently in the real-time community. Much research effort has been shifted from path analysis to microarchitecture modeling.

The CPU pipeline is considered to be relatively easy to model because it is only effected by adjacent instructions. The cache memory system poses a much bigger challenge. Fetching an instruction may result in a cache miss, causing it and perhaps some surrounding instructions to be loaded into the cache line. This action has two effects: (i) subsequent instructions are more likely to be found in the cache, and (ii) the displaced cache contents may later cause cache misses if they are needed again. Therefore, during the program path analysis stage, if any portion of the instruction sequence is altered, the cache memory activities of the *whole* sequence will be affected and a new cache analysis is required. This often leads to explicit enumeration of program paths. Data cache analysis is even more difficult because some data addresses may not be determined statically.

Several WCET analysis with instruction cache modeling methods have been proposed. Liu and Lee [5] note that a *sufficient* condition for determining the *exact* worst case cache behavior is to search through all feasible program paths exhaustively. This becomes an intractable problem whenever there is a conditional statement inside a while loop, which unfortunately happens frequently. Lim *et al.* [6], who extend Shaw's timing schema methodology [7] to incorporate cache analysis, also encounter a similar problem. To deal with this intractable problem, the above researchers trade off cache prediction accuracy for computational complexity by proposing different pessimistic heuristics. Even so, the size of the program for analysis is still limited. Arnold *et al.* [8] propose a less aggressive cache analysis method. They use flow analysis to identify the *potential* cache conflicts and classify each instruction as first miss, always hit, always miss or first hit categories. This results in fast but less accurate cache analysis. Rawat [9] handles data cache performance analysis by using graph-coloring techniques. However, this approach has limited success even for small programs. A severe drawback of all the above methods is that they cannot handle any user annotations describing infeasible program paths, which are essential in tightening the estimated WCET.

Explicit path enumeration is *not* a necessity in obtaining tight estimated WCET. An important observation here is that the WCET can be computed by methods other than path enumeration. We propose a method that determines the worst case *execution counts* of the instructions and from these counts, computes the estimated WCET. The main advantage of this method is that it reduces the solution search space significantly. Further, as we will show in Section 4, only minimal necessary sequencing information is kept in doing the cache analysis. No path enumeration is needed. The method supports user annotations that is at least as powerful as Park's Information Description Language (IDL), and at the same time, computes the cache memory activity that is far more accurate than Lim's work. To the best of our knowledge, our research is the first to address both issues together.

3 Program Path Analysis

In this section, we will show how our method handles the program path analysis problem. Here, we use a simple microarchitecture model that assumes the execution time of an instruction to be a constant, i.e., every instruction fetch is assumed to result in a cache miss. This pessimistic assumption will be removed in Section 4 when we introduce a more sophisticated microarchitecture modeling that includes the cache analysis.

As stated in the previous section, our method uses the counting approach to compute the estimated WCET. The method converts the problem of solving the estimated WCET into a set of integer linear programming (ILP) problems in which the estimated WCET, and the worst case execution counts of the instructions are solved for. There are similarities between our analysis technique and the one used by Avrunin *et al.* [10] in determining time bounds for concurrent systems. More details will be provided in the following subsections.

3.1 ILP Formulation

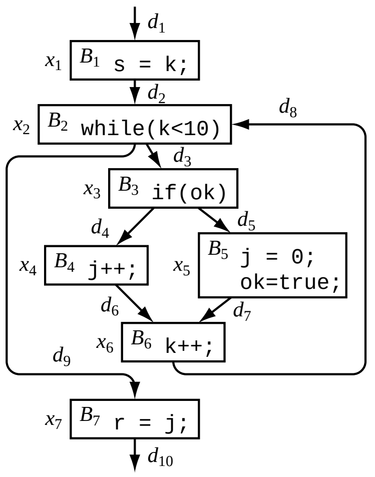
Since we assume that each instruction takes a constant time to execute, the total execution time can be computed by summing the products of instruction counts by their corresponding instruction execution times. Furthermore, since the instructions within a basic block[2] are always executed together, their execution counts are always the same. This allows us to consider them as a single unit. If we let x_i be the execution count of a basic block B_i, and c_i be the execution time of the basic block, then given that there are N basic blocks in the program, the total execution time of the program is given as:

$$\text{Total execution time} = \sum_{i}^{N} c_i x_i. \tag{1}$$

The possible values of x_i's are constrained by the program structure and the possible values of the program variables. If we can represent these constraints as linear inequalities, then the problem of finding the estimated WCET of a program is reduced to an integer linear programming (ILP) problem which can be solved by many existing ILP solvers.

[2]A basic block is a maximum sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

```
/* k >= 0 */
s = k;
while (k < 10) {
  if (ok)
    j++;
  else {
    j = 0;
    ok = true;
  }
  k++;
}
r = j;
```



(i) Code (ii) CFG

Figure 1: An example code fragment showing how the structural and functionality constraints are constructed.

The linear constraints are divided into two parts: (i) **program structural constraints**, which are derived automatically from the program's control flow graph (CFG), and (ii) **program functionality constraints**, which are provided by the user to specify loop bounds and other path information or extracted from the program semantics. The construction of these constraints is best illustrated by an example shown in Fig. 1, in which a conditional statement is nested inside a while loop. Fig. 1(ii) shows the CFG. Each node in the CFG represents a basic block B_i. A basic block execution count, x_i, is associated with each node. Each edge in the CFG is labeled with a variable d_i which serves both as a label for that edge and as a count of the the number of times that the program control passes through that edge. Analysis of the CFG is equivalent to a standard network-flow problem. Structural constraints can be derived from the CFG from the fact that, for each node B_i, its execution count is equal to the number of times that the control enters the node (inflow), and is also equal to the number of times that the control exits the node (outflow). The structural constraints of this example are:

$$d_1 = 1 \tag{2}$$
$$x_1 = d_1 = d_2 \tag{3}$$
$$x_2 = d_2 + d_8 = d_3 + d_9 \tag{4}$$
$$x_3 = d_3 = d_4 + d_5 \tag{5}$$
$$x_4 = d_4 = d_6 \tag{6}$$
$$x_5 = d_5 = d_7 \tag{7}$$
$$x_6 = d_6 + d_7 = d_8 \tag{8}$$
$$x_7 = d_9 = d_{10}. \tag{9}$$

The first constraint (2) is needed to specify that the code fragment is executed once.

The structural constraints do not provide any loop bound information. This information can be provided by the user as a functionality constraint. In this example, we note that since `k` is positive before it enters the loop, the loop body will be executed between 0 and 10 times each time the loop is entered. The constraints to specify this information are:

$$0x_1 \leq x_3 \leq 10x_1, \tag{10}$$

The functionality constraints can also be used to specify other path information. For example, we observe that the `else` statement (B_5) can be executed at most once inside the loop. This information can be specified as:

$$x_5 \leq 1x_1. \tag{11}$$

More complicated path information can also be specified. For instance, the user may know that if the `else` statement is executed, then the loop will be executed exactly 5 times. The constraint to represent this information is:

$$(x_5 = 0) \mid (x_5 \geq 1 \ \& \ x_3 = 5x_1) \tag{12}$$

Here, the symbols '&' and '|' represent conjunction and disjunction respectively. This constraint is not a linear constraint by itself, but a disjunction of linear constraints sets. This can be viewed as a set of constraint sets, where at least one constraint set member must be satisfied. We have been able to show that all language constructs in Park's IDL can be transformed into sets of linear constraints. As a result, using the linear constraints is at least as descriptive as using IDL.

3.2 Solving the Constraints

Because of the '&' and '|' operators, the program functionality constraints may, in general, be a disjunction of conjunctive constraint sets. To solve the estimated WCET, each set of the functionality constraint sets is combined (the conjunction taken) with the set of structural constraints. The combined set is passed to the ILP solver with (1) to be maximized. The ILP solver returns the maximum value of the expression, as well as the basic block counts. The above procedure is repeated for every functionality constraint set. The maximum over all these running times is the estimated WCET.

The total time required to solve the estimated WCET depends on the number of functionality constraint sets and the time to solve each constraint set. Although the number of functionality constraint sets double every time a functionality constraint with disjunction operator '|' is added, we found the size to be small for all the experiments we did as reported in Section 6. The second issue is the complexity of solving each ILP problem, which is, in general, an *NP*-hard problem. We are able to demonstrate that if we restrict our functionality constraints to those that correspond to the constructs in IDL, then the ILP problem collapses to a network flow problem, which can be solved in polynomial time. Our experiments show that the time to solve the estimated WCET is negligible.

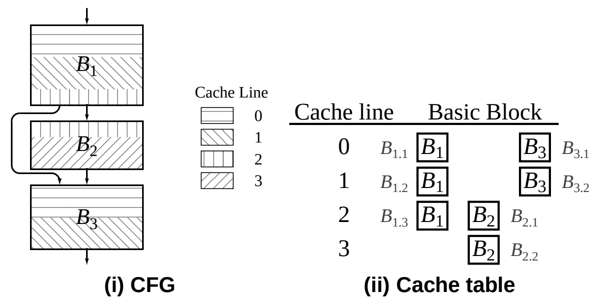

B_1
B_2
B_3
Cache Line
0
1
2
3
Cache line
Basic Block
0 $B_{1.1}$ B_1 B_3 $B_{3.1}$
1 $B_{1.2}$ B_1 B_3 $B_{3.2}$
2 $B_{1.3}$ B_1 B_2 $B_{2.1}$
3 B_2 $B_{2.2}$
(i) CFG
(ii) Cache table

Figure 2: An example showing how the *l-blocks* are constructed. Each rectangle in the cache table represents a *l-block*.

4 Microarchitecture Modeling

Our goal is to model the CPU pipeline and the cache memory systems and find out the execution times (c_i's) of the basic blocks. In this paper, we will limit our method to model a direct-mapped instruction cache. However, it can be extended to handle set associative instruction cache memory.

4.1 Direct-mapped Instruction Cache Analysis

To incorporate cache memory analysis into our ILP model shown in the previous section, we will need to modify the cost function (1) and add a list of linear constraints, denoted as **cache constraints**, representing the cache memory behavior. These will be described in the following subsections.

4.1.1 Modified Cost Function

With cache memory, the execution time of an instruction will be different depending on whether it results in a cache hit or cache miss. Thus, we need to subdivide the original instruction counts into counts of cache hits and misses. If we can determine these counts, and the hit and miss execution times of each instruction, then a tighter bound on the execution time of the program can be established. As in the previous section, we can group adjacent instructions together. We define a new type of atomic structure for analysis, the *line-block* or simply *l-block*. A *l-block* is defined as a contiguous sequence of instructions within the same basic block that are mapped to the same line in the instruction cache. All instructions within an *l-block* will always have the same cache hit/miss counts, and the same total execution counts.

Fig. 2(i) shows a CFG with 3 basic blocks. Suppose that the instruction cache has 4 lines. Since the starting address of each basic block can be determined from the program's executable code, we can find all the cache lines that instructions within it map to, and add an entry on these cache lines in the cache table (Fig. 2(ii)). The boundary of each *l-block* is shown by the solid line rectangle. Suppose a basic block B_i is partitioned into n_i *l-blocks*. We denote these *l-blocks* as $B_{i.1}$, $B_{i.2}$, ..., $B_{i.n_i}$.

For any two *l-blocks* that map to the same cache line, they **conflict** with each other if the execution of one *l-block* will displace the cache content of the other. Otherwise, they are called **non-conflicting** l-blocks (e.g. $B_{1.3}$ and $B_{2.1}$ in Fig. 2).

Since *l-block* $B_{i.j}$ is inside the basic block B_i, its execution count is equal to x_i. The cache hit and the cache miss counts of *l-block* $B_{i.j}$ are denoted as $x_{i.j}^{hit}$ and $x_{i.j}^{miss}$ respectively, and

$$x_i = x_{i.j}^{hit} + x_{i.j}^{miss}, \qquad 1 \le j \le n_i \tag{13}$$

The new total execution time (cost function) is given by:

$$\text{Total execution time} = \sum_{i}^{N} \sum_{j}^{n_i} (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss}). \tag{14}$$

where $c_{i.j}^{hit}$ and $c_{i.j}^{miss}$ are the hit cost and the miss cost of the *l-block* $B_{i.j}$ respectively.

Equation (13) links the new cost function (14) with the program structural constraints and the program functionality constraints, which remain unchanged. In addition, the cache behavior can now be specified in terms of the new variables $x_{i.j}^{hit}$'s and $x_{i.j}^{miss}$'s.

4.1.2 Cache Constraints

These constraints are used to constrain the hit/miss counts of the *l-blocks*. Consider a simple case. For each cache line, if there is only one *l-block* $B_{k.l}$ mapping to it, then once $B_{k.l}$ is loaded into the cache it will permanently stay there. In other words, only the first execution of this *l-block* may cause a cache miss and all subsequent executions will result in cache hits. Thus,

$$x_{k.l}^{miss} \le 1. \tag{15}$$

A slightly more complicated case occurs when two or more **non-conflicting** *l-blocks* map to the same cache line, such as $B_{1.3}$ and $B_{2.1}$ in Fig. 2. The execution of any of them will load all the *l-blocks* into the cache line. Therefore, the sum of their cache miss counts is at most one. In this example, the constraint is:

$$x_{1.3}^{miss} + x_{2.1}^{miss} \le 1. \tag{16}$$

When a cache line contains two or more **conflicting** *l-blocks*, the hit/miss counts of all the *l-blocks* mapped to this line will be affected by the sequence in which these *l-blocks* are executed. An important observation is that the execution of any other *l-blocks* from other cache lines will have no effect on these counts. This leads us to examine the control flow of the *l-blocks* mapped to that particular cache line by defining a *cache conflict graph*.

4.1.3 Cache Conflict Graph

A cache conflict graph (CCG) is constructed for every cache line containing two or more conflicting *l-blocks*. It contains a start node 's', an end node 'e', and a node '$B_{k.l}$' for every *l-block* $B_{k.l}$ mapped to the same cache line. The start node represents the start of the program, and the end node represents the end of the program. For every node '$B_{k.l}$', a directed edge is drawn from node $B_{k.l}$ to node $B_{m.n}$ if there exists a path in the CFG from basic block B_k to basic block B_m *without* passing through the basic blocks of any other *l-blocks* of the same cache line. If there is a path from the start of the CFG to basic block B_k without going through the basic blocks of any other *l-blocks* of the same cache line, then a directed edge is drawn from the start node to node $B_{k.l}$. The edges between nodes and the end node are constructed analogously. Suppose that a cache line contains only two conflicting *l-blocks* $B_{k.l}$ and $B_{m.n}$. A possible CCG is shown in Fig. 3. The program control begins at the start node. After executing some other *l-blocks* from other cache lines, it will eventually reach any one of node $B_{k.l}$, node $B_{m.n}$ or the end node. Similarly, after executing $B_{k.l}$, the control may pass through some *l-blocks* from other cache lines and then reach to node $B_{k.l}$ again or it may reach node $B_{m.n}$ or the end node.

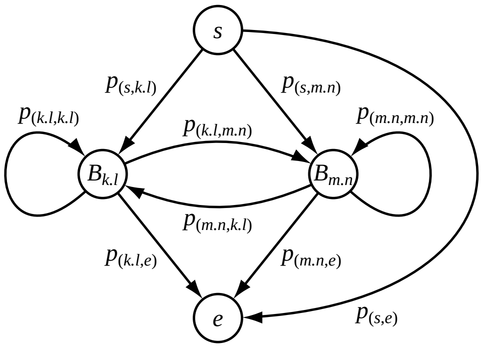

$$\begin{tikzcd}
 & s \arrow[dl, "p_{(s,k.l)}"'] \arrow[dr, "p_{(s,m.n)}"] \arrow[dd, bend left=80, "p_{(s,e)}"] & \\
B_{k.l} \arrow[loop left, "p_{(k.l,k.l)}"] \arrow[rr, bend left, "p_{(k.l,m.n)}"] \arrow[dr, "p_{(k.l,e)}"'] & & B_{m.n} \arrow[loop right, "p_{(m.n,m.n)}"] \arrow[ll, bend left, "p_{(m.n,k.l)}"] \arrow[dl, "p_{(m.n,e)}"] \\
 & e &
\end{tikzcd}$$

Figure 3: A general cache conflict graph containing two conflicting *l-blocks*.

For each edge from node $B_{i.j}$ to node $B_{u.v}$, we assign a variable $p_{(i.j,u.v)}$ to count the number of times that the control passes through that edge. At each node $B_{i.j}$, the sum of control flow going into the node must be equal to the sum of control flow leaving the node, and it must also be equal to the execution count of *l-block* $B_{i.j}$. Therefore, two constraints are constructed at each node $B_{i.j}$:

$$x_i = \sum_{u.v} p_{(u.v,i.j)} = \sum_{u.v} p_{(i.j,u.v)}, \tag{17}$$

where '$u.v$' may also include the start node 's' and the end node 'e'. This set of constraints is linked to the program structural and functionality constraints via the x-variables.

The program is executed once, so at start node:

$$\sum_{u.v} p_{(s,u.v)} = 1. \tag{18}$$

The variable $p_{(i.j,i.j)}$ represents the number of times that the control flows into *l-block* $B_{i.j}$ after executing *l-block* $B_{i.j}$

without entering any other *l-blocks* of the same cache line in between. Therefore, the contents of *l-block* $B_{i.j}$ are still in the cache every time the control follows the edge $(B_{i.j}, B_{i.j})$ to reach node $B_{i.j}$, and it will result in a cache hit. Thus, there will be at least $p_{(i.j,i.j)}$ cache hits for *l-block* $B_{i.j}$. In addition, if both edges $(B_{i.j}, e)$ and $(s, B_{i.j})$ exist, then the contents of $B_{i.j}$ may already be in cache at the beginning of program execution as its content may be left by the previous program execution. Thus, variable $p_{(s,i.j)}$ *may* also be counted as a cache hit. Hence,

$$p_{(i.j,i.j)} \leq x^{hit}_{i.j} \leq p_{(s,i.j)} + p_{(i.j,i.j)}. \tag{19}$$

Otherwise, if any of edges $(s, B_{i.j})$ and $(B_{i.j}, e)$ does not exist, then

$$x^{hit}_{i.j} = p_{(i.j,i.j)}. \tag{20}$$

Equations (15) through (20) are the possible cache constraints for bounding the cache hit/miss counts. These constraints, together with (13), the structural constraints and the functionality constraints, are passed to the ILP solver with the goal of maximizing the cost function (14). Because of the cache information, a tighter estimated WCET will be returned. Further, some path sequencing information can be expressed in terms of p-variables as extra functionality constraints. The CCGs are network flow graphs and thus the cache constraints are typically solved rapidly by the ILP solver. In the worst case, there is one CCG for each cache line.

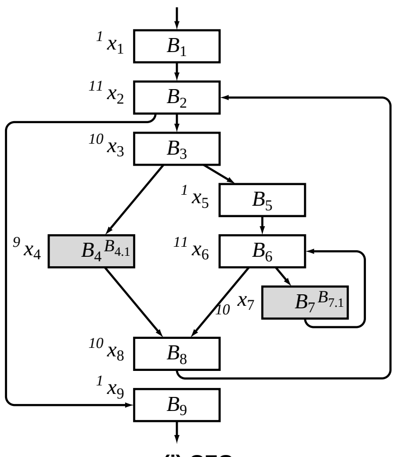
The above constraints can also be used to solve best case execution time. In this case the ILP solver will try to increase the value of $x^{hit}_{i.j}$ as much as possible. If $p_{(i.j,i.j)}$ (self-edge variable) exists, then the ILP solver may set $p_{(i.j,i.j)} = x^{hit}_{i.j} = x_i$. However, this is not possible in any execution trace. Before this path can occur, control must first flow into node $B_{i.j}$ from some other node. To handle this problem, an additional constraint is required for all nodes $B_{i.j}$ with a self-edge:

$$x_i \leq Z \sum_{u.v,\ u.v \neq i.j} p_{(u.v,i.j)}, \tag{21}$$

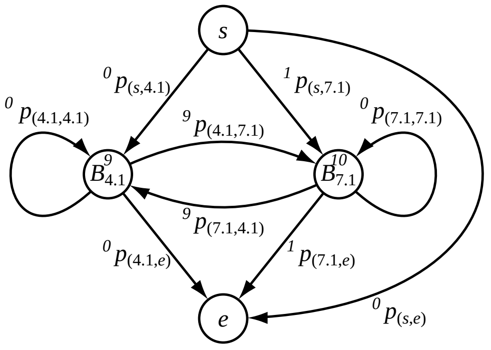
where Z is a large positive integer constant. The addition of this kind of constraints may generate some non-integral optimal variable values when the whole constraint set is passed to LP solver. If the ILP solver uses branch and bound techniques for solving the ILP problem, the computational time may be lengthened significantly.

4.1.4 Bounds on *p*-variables

In this subsection, we discuss bounds on the p-variables. Without the correct bounds, the solver may return an infeasible *l-block* count and an overly pessimistic estimated WCET. This is demonstrated by the example in Fig. 4. In this example, the CFG contains two nested loops. Suppose that there are two conflicting *l-blocks* $B_{4.1}$ and $B_{7.1}$. A CCG will be constructed (Fig. 4(ii)) and the following cache constraints will be generated:



(i) CFG



(ii) CCG

Figure 4: An example showing two conflicting *l-blocks* ($B_{4.1}$ and $B_{7.1}$) from two different loops. The italicized numbers shown on the left of the variables are the pessimistic worst case solution returned from ILP solver.

$$\begin{aligned} x_4 &= p_{(s,4.1)} + p_{(4.1,4.1)} + p_{(7.1,4.1)} \\ &= p_{(4.1,e)} + p_{(4.1,4.1)} + p_{(4.1,7.1)} \end{aligned} \tag{22}$$

$$\begin{aligned} x_7 &= p_{(s,7.1)} + p_{(7.1,7.1)} + p_{(4.1,7.1)} \\ &= p_{(7.1,e)} + p_{(7.1,7.1)} + p_{(7.1,4.1)} \end{aligned} \tag{23}$$

$$p_{(s,4.1)} + p_{(s,7.1)} + p_{(s,e)} = 1 \tag{24}$$

$$p_{(4.1,4.1)} \leq x^{hit}_4 \leq p_{(s,4.1)} + p_{(4.1,4.1)} \tag{25}$$

$$p_{(7.1,7.1)} \leq x^{hit}_7 \leq p_{(s,7.1)} + p_{(7.1,7.1)}. \tag{26}$$

Suppose that the user specifies that both loops will be executed 10 times each time they are entered and that basic block B_4 will be executed 9 times each time the outer loop is entered. The functionality constraints for this information are:

$$x_3 = 10x_1, \quad x_7 = 10x_5, \quad x_4 = 9x_1. \tag{27–29}$$

If we feed the above constraints and the structural constraints into the ILP solver, it will return a worst case solution in which the counts are as shown on the left of the variables in the figure.

From the CCG, we observe that these p-values imply that *l-blocks* $B_{4.1}$ and $B_{7.1}$ will be executed alternately, with *l-block* $B_{7.1}$ being executed first. This execution sequence will generate the maximum number of cache misses and hence the WCET. However, if we look at the CFG, we know that this sequence is impossible because the inner loop will be entered only once. Once the program control enters the inner loop, *l-block* $B_{7.1}$ must be executed 10 times before control exits the inner loop. Hence, there must be at least 9 cache hits for *l-block* $B_{7.1}$. The ILP solver over-estimates the number of cache misses based on the given constraints. Upon closer investigation, we find that the correct solution also satisfies the above set of constraints. This implies that some constraints for tightening the solution space are missing.

The reason for producing such pessimistic worst case solution is that the p-variables are not properly bounded. When we assign the p-variables to the edges of the CCG, we do not specify any upper limits on these p-variables. However, the flow equations (17) place a bound on them. For any variable $p_{(i.j,u.v)}$, its bounds are:

$$0 \le p_{(i.j,u.v)} \le \min(x_i, x_u). \tag{30}$$

Consider the case that two conflicting *l-blocks* $B_{i.j}$ and $B_{u.v}$ are in the same loop and at the same loop nesting level. In this case the maximum control flow allowed between these two *l-blocks* is equal to the total number of loop iterations. This will be the upper bound on $p_{(i.j,u.v)}$. Since *l-blocks* $B_{i.j}$ and $B_{u.v}$ are inside the loop, x_i and x_u can at most be equal to the total number of loop iterations. Therefore, (17) will bound $p_{(i.j,u.v)}$ correctly.

Suppose that there are two nested loops such that *l-block* $B_{i.j}$ is in the outer loop while $B_{u.v}$ is in the inner loop. If edge $(B_{i.j}, B_{u.v})$ exists, all paths represented by this edge go from basic block B_i to basic block B_u in the CFG. They must pass through the loop preheader[3], say basic block B_h, of the inner loop. Since the execution count of basic block B_h, x_h, may be smaller than x_i and x_u, a constraint

$$p_{(i.j,u.v)} \le x_h \tag{31}$$

is needed to properly bound $p_{(i.j,u.v)}$.

In general, a constraint is constructed at each loop preheader. All the paths that go from outside the loop to inside the loop must pass through the loop preheader. Therefore, the sum of these flows can at most be equal to the execution count of the loop preheader. In our example, a constraint at loop preheader B_5 is needed:

$$p_{(s,7.1)} + p_{(4.1,7.1)} \le x_5. \tag{32}$$

[3]A loop preheader is the basic block just before entering the loop. For instance, in the example shown in Fig. 4, basic block B_1 is the loop preheader of the outer loop and basic block B_5 is the loop preheader of the inner loop.

With this constraint, the ILP solver will generate a correct solution.

4.2 Interprocedural Calls

A function may be called many times from different locations of the program. The variable x_i represents the *total* execution count of the basic block B_i when the whole program is executed once. Similarly, $x_{i.j}^{hit}$ and $x_{i.j}^{miss}$ represents the total hit and miss counts of the *l-block* $B_{i.j}$ respectively. Equation (13) is still valid and (14) still represents the total execution time of the program.

Every function call is treated as if it is inlined. During the construction of CFG, a function call is represented by an f-edge pointing to an instance of the callee function's CFG. The edge has a variable f_k which represents the number of times that the particular instance of the callee function is called. Each variable and name in the callee function has a suffix "$.f_k$" to distinguish it from other instances of the same callee function.

Consider the example shown in Fig. 5. Here, function `inc` is called twice in the `main` function. The CFG is shown in Fig. 5(ii). The structural constraints are:

$$d_1 = 1 \tag{33}$$
$$x_1 = d_1 = f_1 \tag{34}$$
$$x_2 = f_1 = f_2 \tag{35}$$
$$d_2.f_1 = f_1 \tag{36}$$
$$x_3.f_1 = d_2.f_1 = d_3.f_1 \tag{37}$$
$$d_2.f_2 = f_2 \tag{38}$$
$$x_3.f_2 = d_2.f_2 = d_3.f_2 \tag{39}$$
$$x_3 = x_3.f_1 + x_3.f_2 \tag{40}$$

The last equation above links the total execution counts of basic block B_3 with its counts from two instances of the function. Based on these variables, the user can provide specific information on different instances of the same function.

The CCG is constructed as before by treating each instance of *l-block* $B_{i.j}.f_k$ as different from other instances of the same *l-block*. In the example, if *l-block* $B_{1.1}$ conflicts with *l-block* $B_{3.1}$, then since *l-block* $B_{3.1}$ has two instances ($B_{3.1}.f_1$ and $B_{3.1}.f_2$), there will be 5 nodes in the CCG (Fig. 5(iii)).

The cache constraints and the bounds on p variables are constructed as before, except the hit constraints are modified slightly. In addition to the self edges, the edge going from one instance of a *l-block* (say $B_{i.j}.f_k$) to another instance of the same *l-block* ($B_{i.j}.f_l$) are counted as the cache hit of the *l-block* $B_{i.j}$, as it represents the execution of *l-block* $B_{i.j}$ at f_l after the same *l-block* has just been executed at f_k. The complete cache constraints derived from the example's CCG are:

$$x_1 = x_{1.1}^{hit} + x_{1.1}^{miss} \tag{41}$$
$$x_2 = x_{2.1}^{hit} + x_{2.1}^{miss} \tag{42}$$

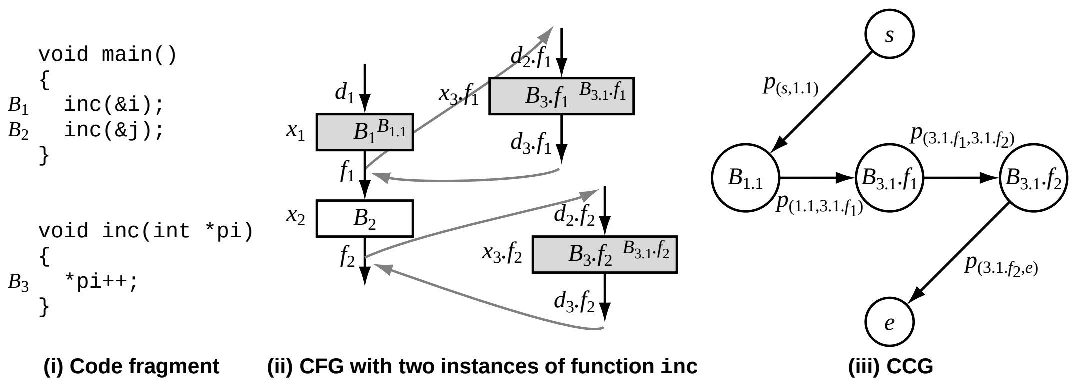


Figure 5: An example code fragment showing how function calls are handled.

$$x_3 = x_{3.1}^{hit} + x_{3.1}^{miss} \tag{43}$$
$$x_{2.1}^{miss} \le 1 \tag{44}$$
$$x_1 = p_{(s,1.1)} = p_{(1.1,3.1.f_1)} \tag{45}$$
$$x_3.f_1 = p_{(1.1,3.1.f_1)} = p_{(3.1.f_1,3.1.f_2)} \tag{46}$$
$$x_3.f_2 = p_{(3.1.f_1,3.1.f_2)} = p_{(3.1.f_2,e)} \tag{47}$$
$$p_{(s,1.1)} = 1 \tag{48}$$
$$x_{1.1}^{hit} = 0 \tag{49}$$
$$x_{3.1}^{hit} = p_{(3.1.f_1,3.1.f_2)}. \tag{50}$$

4.3 CPU Pipeline

Since $c_{i.j}^{hit}$'s and $c_{i.j}^{miss}$'s must be constants, we assume that the time required to execute a sequence of instructions in the CPU pipeline is always a constant throughout the execution of the program. The hit cost $c_{i.j}^{hit}$ of a *l-block* $B_{i.j}$ is determined by adding up the effective execution times of the instructions in the *l-block*. Since the effective execution times of some instructions, especially the the floating point instructions, are data dependent, a conservative approach is taken by assuming the worst case effective execution time. This may induce some pessimism in the final estimated WCET. Additional time is also added to the last *l-block* of each basic block so as to ensure that all the buffered load/store instructions are completed when the control reaches the end of the basic block. The miss cost $c_{i.j}^{miss}$ of the *l-block* is equal to the time needed to load the instructions of the *l-block* into the cache memory and to execute them in the CPU.

5 Implementation

The above cache analysis method has been implemented in a tool called `cinderella`[4], which estimates the WCET of programs running on an Intel QT960 development board [11] containing an 20MHz Intel i960KB processor, 128KB of main memory and several I/O peripherals. The i960KB processor is a 32bit RISC processor used in many embedded systems (e.g. in laser printers). It contains an on-chip 512 byte direct-mapped instruction cache which is organized as 32 × 16-byte lines. It also features a floating point unit, a 4-stage instruction pipeline, and 4 register windows for faster execution of function call instructions [12, 13].

`Cinderella` contains about 15,000 lines of C++ code. The tool reads the subject program's executable code and constructs the CFGs and the CCGs. It then outputs the annotation files in which the x's and f's are labeled along with the program's source code. The user is then asked to provide loop bounds. An estimated WCET can thus be computed. The user can provide additional path information, if available, to tighten this bound. We use a public domain ILP solver `lp_solve`[5] to solve the constraints generated by `cinderella`. The solver uses the branch and bound procedure to solve the ILP problem.

An optimization implemented in `cinderella` actually reduces the number of variables and CCGs. If two or more cache lines can hold instructions from the same set of basic blocks, e.g. cache lines 0 and 1 in Fig. 2(ii), then the corresponding *l-blocks* can be combined and only one CCG is constructed for these cache lines. This technique is used in `cinderella` to improve efficiency.

6 Experimental Results

Our goal is to find a tight bound on a program's WCET. A small amount of pessimism is normally present in the estimated bound. This is due to two factors: (i) insufficient path information from the user so that some infeasible program paths are considered, and (ii) inaccuracy in microarchitecture modeling which affects the accuracy of the values of $c_{i.j}^{hit}$'s and $c_{i.j}^{miss}$'s in (14). The first factor can be reduced by providing more path information and the second can be reduced by a more sophisticated hardware model.

[4]Details of this tool can be obtained via Cinderella WWW home page (`http://www.princeton.edu/~yauli/cinderella/`).

[5]`lp_solve` is written by Michel Berkelaar and can be retrieved at `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

Table 1: Set of Benchmark Examples, their descriptions, source file line size and the binary executable code size.

Function	Description	Lines	Bytes
`check_data`	Example from Park's thesis [4]	23	88
`piksrt`	Insertion Sort	19	104
`sort`	Bubble sort from [8]	41	152
`matcnt`	Summation of 2 matrices, from [8]	85	460
`matcnt2`	`Matcnt` with inlined functions	73	400
`stats`	Calculate the sum, mean and variance of two arrays, from [8]	100	656
`fft`	Fast Fourier Transform	57	500
`jpeg_fdct_islow`	JPEG forward discrete cos transform	300	996
`line`	Line drawing routine from Gupta [14]	165	1,556
`circle`	Circle drawing routine from Gupta	100	1,588
`des`	Data Encryption Standard	192	1,852
`whetstone`	Whetstone benchmark	196	2,760
`dhry`	Dhrystone benchmark	761	1,360

Table 2: Estimated WCETs of Benchmark programs. All values are in units of clock cycles.

Function	Measured WCET	Estimated WCET with cache analysis	Estimated WCET w/o cache analysis
`check_data`	4.30×10^2	4.91×10^2	11.9×10^2
`piksrt`	1.71×10^3	1.74×10^3	5.86×10^3
`sort`	9.99×10^6	27.8×10^6	50.2×10^6
`matcnt`	2.20×10^6	5.46×10^6	8.17×10^6
`matcnt2`	1.86×10^6	2.11×10^6	4.46×10^6
`stats`	1.16×10^6	2.21×10^6	2.95×10^6
`fft`	2.20×10^6	2.63×10^6	3.97×10^6
`jpeg_fdct_islow`	9.05×10^3	9.11×10^3	16.7×10^3
`line`	4.84×10^3	6.09×10^3	9.15×10^3
`circle`	1.45×10^4	1.54×10^4	1.59×10^4
`des`	2.44×10^5	3.70×10^5	6.72×10^5
`whetstone`	6.94×10^6	10.5×10^6	14.9×10^6
`dhry`	5.76×10^5	7.57×10^5	13.3×10^5

In this section, we would like evaluate the accuracy of our cache analysis method as well as examine its performance issues. Since there are no standard benchmark programs, we have selected the benchmark programs from a variety of sources. They include programs from academic sources, DSP applications, and other standard software benchmarks. Table 1 shows the program names, brief descriptions, the size of the source code in number of lines and the binary code size of the program in number of bytes.

Since it is impractical to simulate all the possible program input data and all initial system states, a program's *actual* WCET cannot be computed. Instead, we try to identify the worst case data set by a careful study of the program and use the logic analyzer to measure the program's execution time for this worst case data set. We denote this time as the program's *measured* WCET. A program's measured WCET is always bounded by its actual WCET and we assume that it is very close to the actual WCET.

Table 2 shows the results of our experiments. The second and third columns show the measured WCET and the estimated WCET with cache analysis. For comparison, we also estimate WCET *without* performing the cache analysis. This is shown in the last column. Clearly the WCET bound with cache analysis is much tighter than the one without it. For small programs (e.g. `check_data` and `piksrt`), the estimated WCETs are very close to their corresponding measured WCETs. For larger programs, the differences are larger. We found that this discrepancy is mainly due to two factors. The first is that we assume that the register window overflows (underflows) on each function call (return). This pessimism incurs about 50 clock cycles on each function call and function return. This can be illustrated in examples `matcnt` and `matcnt2`. `Matcnt` has two small functions which are called frequently inside the loops. These function calls generate a large amount of pessimism. In `matcnt2`, these two functions are inlined and the estimated WCET is tightened significantly. We are currently working on this area to reduce the pessimism. The second factor is due to the pessimism in the execution times of floating point instructions. The execution time of a floating point instruction depends on the values of its arguments and its worst case execution time are typically 30%–40% more than its average execution time.

The structural constraints and the cache constraints are derived from the CFG and the CCGs which are very similar to network flow graphs. We therefore expect that the ILP solver can solve the problem efficiently. Table 3 shows, for each program, the number of variables and constraints, the number of branches in solving the ILP problem, and the CPU time required to solve the problem. Since each program may have more than one set of functionality constraints, a '+' symbol is used to separate the number of functionality constraints in each set. For a program having *n* sets of functionality constraints, the ILP will be called *n* times. The '+' symbol is once again used to separate the number of ILP branches and the CPU time for each ILP call.

We found that even with thousands of variables and constraints, the branch and bound ILP solver can still find an integer solution within the first few calls to the linear programming solver. The time taken to solve the problem ranges from less than a second to a few minutes on a SGI Indigo^2 workstation. With a commercial ILP solver CPLEX, the CPU time reduces significantly to a few seconds.

In order to evaluate how the cache size will effect the time needed for solving the problem, we double the number of cache lines (and hence the cache size) from 32 lines to 64 lines and find the CPU time needed to solve the problems. Table 4 shows the results. From the table, we find that the number of variables and the number of constraints change little when the number of cache lines is doubled. The solution time is of the same order as before. The primary reason is that although increasing the number of cache lines will increase the number of CCGs and hence more cache constraints are generated, each CCG has fewer nodes and edges. As a result, there are fewer cache constraints in each CCG. These two factors roughly cancel out each other.

Table 3: Performance issues in cache analysis.

Function	No. of Variables				No. of Constraints			ILP	Time
	d's	*f*'s	*p*'s	*x*'s	Struct.	Cache	Funct.	branches	(sec.)
`check_data`	12	0	0	40	25	21	5+5	1+1	0+0
`piksrt`	12	0	0	42	22	26	4	1	0
`sort`	15	1	0	58	35	31	6	1	0
`matcnt`	20	4	0	106	59	61	4	1	0
`matcnt2`	20	2	0	92	49	54	4	1	0
`stats`	28	13	75	180	99	203	4	1	0
`fft`	27	0	0	80	46	46	11	1	0
`jpeg_fdct_islow`	8	0	18	34	16	49	2	1	0
`line`	31	2	264	231	73	450	2	1	3
`circle`	8	1	81	100	24	186	1	1	0
`des`	174	11	728	560	342	1,059	16+16	13+13	171+197
`whetstone`	52	3	301	388	108	739	14	1	2
`dhry`	102	21	503	504	289	777	$24\times4+26\times4$	1×8	$0\times3+2+0+1\times2+4$

Table 4: The complexity of the ILP problem when the number of cache lines is doubled to 64 lines.

Function	No. of variables				No. of constraints			ILP	Time
	d's	*f*'s	*p*'s	*x*'s	Struct.	Cache	Funct.	branches	(sec.)
`des`	174	11	809	524	342	1,013	16+16	7+10	90+145
`whetstone`	52	3	232	306	108	559	14	1	1

7 Conclusions and Future Work

In this paper, we present a method to determine a tight bound on a program's worst case execution time. The method includes a direct-mapped instruction cache analysis and uses an integer linear programming formulation to solve the problem. This approach avoids enumeration of program paths. Furthermore, it allows the user to provide program path annotations so that a tighter bound may be obtained. The method is implemented in the tool `cinderella` and the experimental results show that the WCET bound is much closer to the measured WCET than if cache analysis is not included. Since the linear constraints are mostly derived from the network flow graphs, the ILP problems are typically solved efficiently.

We are now working on set-associative instruction cache and data cache memory modeling, as well as the register window modeling.

References

[1] Eugene Kligerman and Alexander D. Stoyenko, "Real-time Euclid: A language for reliable real-time systems", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 941–949, September 1986.

[2] P. Puschner and Ch. Koza, "Calculating the maximum execution time of real-time programs", *The Journal of Real-Time Systems*, vol. 1, no. 2, pp. 160–176, September 1989.

[3] Aloysius K. Mok, Prasanna Amerasinghe, Moyer Chen, and Kamtron Tantisirivat, "Evaluating tight execution time bounds of programs by annotations", in *Proceedings of the 6th IEEE Workshop on Real-Time Operating Systems and Software*, May 1989, pp. 74–80.

[4] Chang Yun Park, *Predicting Deterministic Execution Times of Real-Time Programs*, PhD thesis, University of Washington, Seattle 98195, August 1992.

[5] Jyh-Charn Liu and Hung-Ju Lee, "Deterministic upperbounds of the worst-case execution times of cached programs", in *Proceedings of the 15th IEEE Real-Time Systems Symposium*, December 1994, pp. 182–191.

[6] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, and Chong Sang Kim, "An accurate worst case timing analysis technique for RISC processors", in *Proceedings of the 15th IEEE Real-Time Systems Symposium*, December 1994, pp. 97–108.

[7] Alan C. Shaw, "Reasoning about time in higher-level language software", *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 875–889, July 1989.

[8] Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon, "Bounding worst-case instruction cache performance", in *Proceedings of the 15th IEEE Real-Time Systems Symposium*, December 1994, pp. 172–181.

[9] Jai Rawat, "Static analysis of cache performance for real-time programming", Master's thesis, Iowa State University of Science and Technology, November 1993, TR93-19.

[10] George S. Avrunin, James C. Corbett, Laura K. Dillon, and Jack C. Wileden, "Automated derivation of time bounds in uniprocessor concurrent systems", *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 708–719, September 1994.

[11] Intel Corporation, *QT960 User Manual*, 1990, Order Number 270875-001.

[12] Intel Corporation, *i960KA/KB Microprocessor Programmers's Reference Manual*, 1991, ISBN 1-55512-137-3.

[13] Glenford J. Myers and David L. Budde, *The 80960 Microprocessor Architecture*, John Wiley & Sons, Inc., 1988, ISBN 0-471-61857-8.

[14] Rajesh Kumar Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, PhD thesis, Stanford University, December 1993.