Performance Estimation of Embedded Software with Instruction Cache Modeling

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

Embedded systems generally interact with the outside world. Thus, some real-time constraints may be imposed on the system design. Verification of these constraints requires computing a tight upper bound on the worst case execution time (WCET) of a hardware/software system. The problem of bounding WCET is particularly difficult on modern processors, which use cache-based memory systems that vary memory access time significantly. This must be accurately modeled in order to tightly bound WCET. Existing approaches either search all possible program paths, an intractable problem, or they use pessimistic assumptions to limit the search space. In this paper we present a far more effective and accurate method for modeling instruction cache activity and computing a tight bound on WCET. It is implemented in the program `cinderella`*. We present some preliminary results of using this tool on sample embedded programs.*

1 Introduction

The execution time of a program often can vary significantly from one run to the next on the same system. In many cases it is essential to know the worst case execution time (WCET) for a hardware/software system. In hard real-time systems, the programmer must guarantee that the WCET satisfies the timing deadlines. Many real-time operating systems rely on this data for process scheduling. In embedded system designs, the WCET of the software is often required for deciding how hardware/software partitioning is done.

Since it is impractical to simulate all possible input data and initial system states, an upper bound on WCET, denoted as the *estimated WCET*, is usually computed by static analysis of the program. WCET estimates are particularly difficult for modern microprocessors, which usually include pipelined instruction execution and cache-based memory systems. These features speed up the typical performance of the system, but complicate worst case timing analysis. The exact execution time of an instruction may vary significantly. The effect of pipelining on execution time can be found relatively easily and accurately since it is affected only by adjacent instructions. The cache memory system poses a much bigger challenge. To accurately tell whether or not the execution of an instruction results in cache hit, a global analysis of the program is required.

Incorporating accurate cache modeling into worst case timing analysis is essential in order to compute *tight* estimated WCETs of programs running on modern processors. In this paper, we propose a method to model direct-mapped instruction caches. Unlike other cache analysis methods, it does not impose any pessimistic assumptions on the cache activity and it yields an accurate solution in an effective way. This method is integrated into our previous work [1] on program path analysis for determining the WCET.

2 Related Work

The problem of determining a program's estimated WCET is in general undecidable and is equivalent to a halting problem. The sufficient conditions for it to be decidable are: (i) no recursive function calls, (ii) no dynamic structures and (iii) bounded loops [2].

Several WCET analysis with instruction cache modeling methods have been proposed. Liu and Lee [3] 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.* [4], who extend Shaw's timing schema methodology [5] to incorporate cache analysis, also encountered 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. Arnold *et al.* [6] 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 [7] handles data cache performance analysis by using graph-coloring techniques. However, this approach has limited success even for small programs. All the above methods encounter computational complexity because they try to determine the *exact sequence* of cache hits and misses. Different pessimistic methods are thus proposed to cope with this complexity, and they result in loose estimated WCET. Another severe drawback is that they cannot handle any user annotations describing infeasible program paths, which are essential in tightening the estimated WCET.

Our objective is to determine tight estimated WCETs, *not* the exact cache activity. The key observation is that the estimated WCET is only affected by the numbers of cache hits and cache misses. The actual sequence of hits and misses have no effect on the estimated WCET. An important aspect to cope with the complexity is to capture only the necessary information that effects the estimated WCET, and ignore all other information as much as possible. Based on this observation, we extend our previous work on worst case path analysis, which uses integer linear programming (ILP) for estimating WCET.

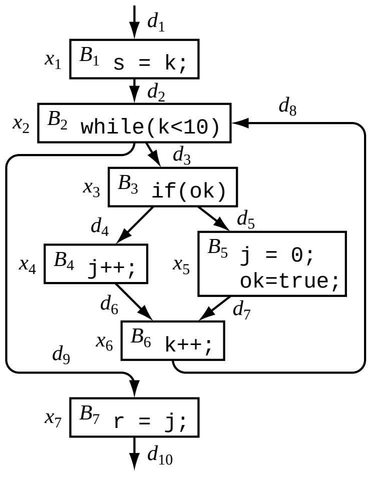
3 ILP Formulation

Our previous work on WCET estimation [1] is focused on path analysis given a simple microarchitectural model. This model assumes that every instruction takes a constant time to execute. Instead of searching all program paths, our strategy is to analytically find the dynamic execution count of each instruction under a worst case scenario. Given that each instruction takes a constant time to execute, the total execution time can be computed by summing the product of instruction counts by their corresponding instruction execution times. Since all instructions within a basic block must have the same execution counts, they can be considered 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, the total execution time of the program is given as:

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

The possible values of x_i 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 will become an ILP problem.

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



(i) Code **(ii) CFG**

Fig. 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 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. The construction of these constraints is best illustrated by an example shown in Fig. 1, which shows a conditional statement nested inside a while loop. Each node in the CFG represents a basic block B_i. A basic block execution count, x_i, is associated with it. Each edge in the CFG is labeled with a variable d_i which counts the the number of times that the program control passes through that edge. Structural constraints are derived from the CFG from the fact that, for each node B_i, its execution count is equal to the sum of control inflow and also the sum of control outflow. The structural constraints of this example are:

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

Here, the first constraint (2) is the starting condition that specifies the code fragment is to be executed once.

The loop bound information is provided by the user as a functionality constraint. In this example, since variable `k` is positive before the control enters the loop, the loop body will be iterated between 0 and 10 times each time it is entered. The constraints to specify this information are:

$$0x_1 \le x_3 \le 10x_1, \quad (10)$$

The functionality constraints can also be used to specify other path information: e.g., the else statement (B_5) can be executed at most once inside the loop:

$$x_5 \le 1x_1. \quad (11)$$

All these constraints ((2) through (11)) are passed to the ILP solver with the goal of maximizing cost function (1). The ILP solver will return the estimated WCET and the worst case values of the variables.

4 Instruction Cache Analysis

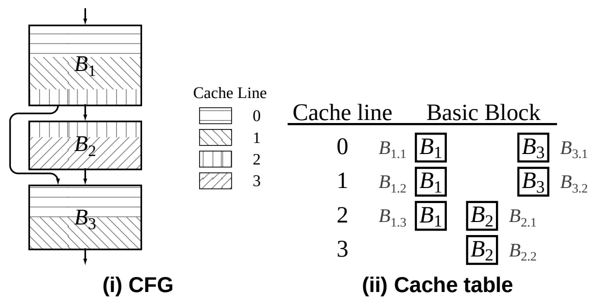
Our goal is to incorporate instruction cache memory analysis. In this paper we will restrict ourselves to direct-mapped caches, however, this method can be extended to set associative instruction caches. We would like to include this analysis into our ILP model shown in the previous section without changing the established linear constraints. This is accomplished by modifying the cost function (1) and by adding a set of linear **cache constraints** representing the cache memory behavior. These will be described in the following subsections.

4.1 New Cost Function

With cache memory, the execution time of an instruction may 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 find 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 the 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. For each basic block, we 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).



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

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

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

$$x_i = x_{i.j}^{hit} + x_{i.j}^{miss}, \qquad 1 \le j \le n_i \quad (12)$$

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

$$\text{Total execution time} = \sum_{i}^{N} \sum_{j}^{n_i} (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss}). \quad (13)$$

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 (12) links the new cost function (13) 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.

4.2 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,

$$x_{k.l}^{miss} \le 1. \quad (14)$$

A slightly more complicated case occurs when two or more **non-conflicting** *l-blocks* map to the same cache line. 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:

$$x_{1.3}^{miss} + x_{2.1}^{miss} \le 1. \quad (15)$$

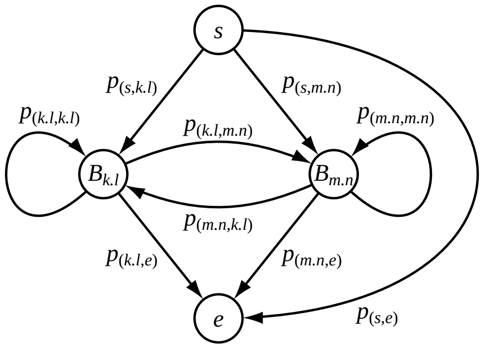


Fig. 3: A general cache conflict graph containing two conflicting *l-blocks*.

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 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*.

4.3 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.

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 $B_{i.j}$:

$$x_i = \sum_{u.v} p_{(u.v,i.j)} = \sum_{u.v} p_{(i.j,u.v)}, \qquad (16)$$

where '$u.v$' may also include the start node 's' and the end node 'e'. Note that this set of constraints links the p-variables to the program structural and functionality constraints via the x-variables.

The program is executed once, so at start node:

$$\sum_{u.v} p_{(s,u.v)} = 1. \qquad (17)$$

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,

$$p_{(i.j,i.j)} \le x^{hit}_{i.j} \le p_{(s,i.j)} + p_{(i.j,i.j)}. \qquad (18)$$

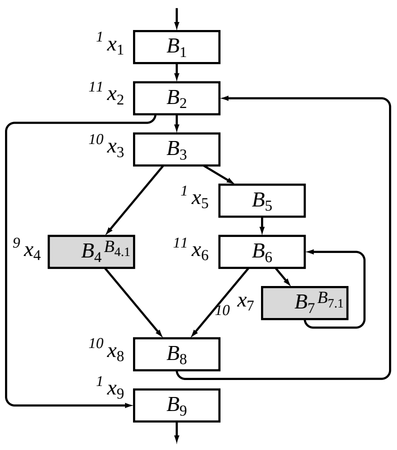
Otherwise, if any of edges $(s, B_{i.j})$ and $(B_{i.j}, e)$ does not exist, then

$$x^{hit}_{i.j} = p_{(i.j,i.j)}. \qquad (19)$$

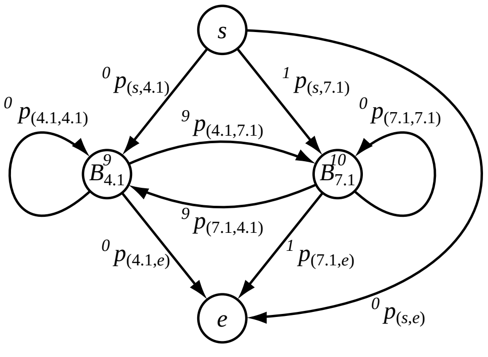
Equations (14) through (19) are the possible cache constraints for bounding the cache hit/miss counts. These constraints, together with (12), the structural constraints and the functionality constraints, are passed to the ILP solver with the goal of maximizing the cost function (13). Because of the cache 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.

4.4 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

Fig. 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.

$$\begin{aligned} x_4 &= p_{(s,4.1)} + p_{(4.1,4.1)} + p_{(7.1,4.1)} \\ &= p_{(4.1,e)} + p_{(4.1,4.1)} + p_{(4.1,7.1)} \qquad (20) \\ x_7 &= p_{(s,7.1)} + p_{(7.1,7.1)} + p_{(4.1,7.1)} \\ &= p_{(7.1,e)} + p_{(7.1,7.1)} + p_{(7.1,4.1)} \qquad (21) \end{aligned}$$

$$p_{(s,4.1)} + p_{(s,7.1)} + p_{(s,e)} = 1 \qquad (22)$$

$$p_{(4.1,4.1)} \leq x_4^{hit} \leq p_{(s,4.1)} + p_{(4.1,4.1)} \qquad (23)$$

$$p_{(7.1,7.1)} \leq x_7^{hit} \leq p_{(s,7.1)} + p_{(7.1,7.1)}. \qquad (24)$$

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:

$$x_3 = 10x_1, \quad x_7 = 10x_5, \quad x_4 = 9x_1. \qquad (25–27)$$

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 of these p-variables. However, the flow equations (16) place a bound on them. For any variable $p_{(i.j,u.v)}$, its bounds are:

$$0 \leq p_{(i.j,u.v)} \leq \min(x_i, x_u). \qquad (28)$$

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, (16) 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[1], 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

$$p_{(i.j,u.v)} \leq x_h \qquad (29)$$

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:

$$p_{(s,7.1)} + p_{(4.1,7.1)} \leq x_5. \qquad (30)$$

[1]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.

5 Interprocedural Calls

So far our cache analysis discussion has been limited to a single function. In this section, we show how function calls are handled.

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 (12) is still valid and (13) 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:

$$d_1 = 1 \tag{31}$$
$$x_1 = d_1 = f_1 \tag{32}$$
$$x_2 = f_1 = f_2 \tag{33}$$
$$d_2.f_1 = f_1 \tag{34}$$
$$x_3.f_1 = d_2.f_1 = d_3.f_1 \tag{35}$$
$$d_2.f_2 = f_2 \tag{36}$$
$$x_3.f_2 = d_2.f_2 = d_3.f_2 \tag{37}$$
$$x_3 = x_3.f_1 + x_3.f_2 \tag{38}$$

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:

$$x_1 = x_{1.1}^{hit} + x_{1.1}^{miss} \tag{39}$$
$$x_2 = x_{2.1}^{hit} + x_{2.1}^{miss} \tag{40}$$
$$x_3 = x_{3.1}^{hit} + x_{3.1}^{miss} \tag{41}$$
$$x_{2.1}^{miss} \leq 1 \tag{42}$$
$$x_1 = p_{(s,1.1)} = p_{(1.1,3.1.f_1)} \tag{43}$$
$$x_3.f_1 = p_{(1.1,3.1.f_1)} = p_{(3.1.f_1,3.1.f_2)} \tag{44}$$
$$x_3.f_2 = p_{(3.1.f_1,3.1.f_2)} = p_{(3.1.f_2,e)} \tag{45}$$
$$p_{(s,1.1)} = 1 \tag{46}$$
$$x_{1.1}^{hit} = 0 \tag{47}$$
$$x_{3.1}^{hit} = p_{(3.1.f_1,3.1.f_2)}. \tag{48}$$

6 Implementation & Hardware Modeling

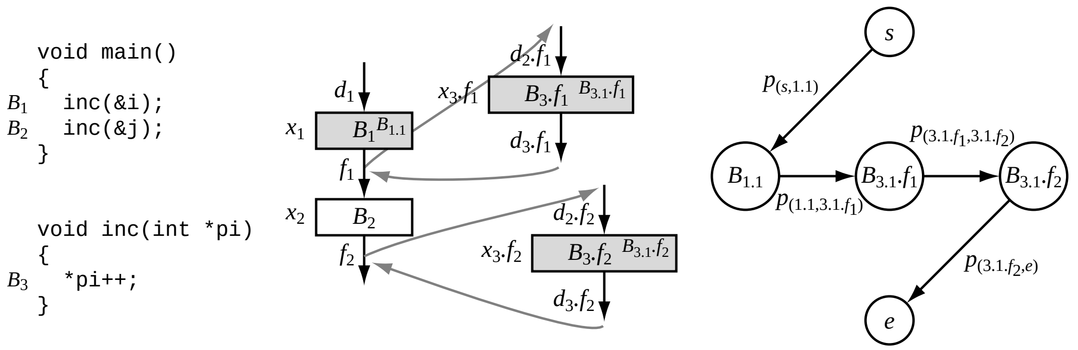
The above cache analysis method has been included into our original tool `cinderella`, which estimates the WCET of programs running on an Intel QT960 development board [8] containing an 20MHz Intel i960KB processor, 128KB of main memory and several I/O peripherals. The i960KB processor is a 32bit RISC processor used in many embedded systems (e.g. in laser printers). It contains an on-chip 512 byte direct-mapped instruction cache organized as 32×16-byte lines. It also features a floating point unit, a 4-stage instruction pipeline, and 4 register windows [9, 10].

The hit cost $c_{i.j}^{hit}$ of a *l-block* $B_{i.j}$ is found by adding up the *worst case* effective execution times of the instructions in the *l-block*. This may induce some pessimism in $c_{i.j}^{hit}$. 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 exits the basic block. The miss cost $c_{i.j}^{miss}$ is equal to the hit cost $c_{i.j}^{hit}$ plus the cache miss penalty.

`Cinderella` contains about 15,000 lines of C++ code. It 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 labelled along with the program's source code. The user is then asked to provide loop bounds. A WCET bound can thus be computed and additional path information can be provided. The constraints are solved by the public domain ILP solver `lp_solve`[2], which uses the branch and bound procedure to solve ILP problems.

An optimization implemented in `cinderella` actually reduces the number of CCGs and variables. If two or

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



```
   void main()
   {
B1   inc(&i);
B2   inc(&j);
   }

   void inc(int *pi)
   {
B3   *pi++;
   }
```

(i) Code fragment **(ii) CFG with two instances of function inc** **(iii) CCG**

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

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 [11]	17	88
`piksrt`	Insertion Sort	15	80
`line`	Line drawing routine in Gupta's thesis [12]	143	1,556
`circle`	Circle drawing routine in Gupta's thesis	88	1,588
`fft`	Fast Fourier Transform	56	544
`des`	Data Encryption Standard	185	1,796
`fullsearch`	MPEG2 encoder frame search routine	204	1,436
`whetstone`	Whetstone benchmark	245	2,760
`dhry`	Dhrystone benchmark	480	1,360
`matgen`	Matrix routine in Linpack benchmark	50	248

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`	4.41×10^2	4.91×10^2	11.9×10^2
`piksrt`	1.79×10^3	1.82×10^3	5.01×10^3
`line`	4.85×10^3	6.09×10^3	9.15×10^3
`circle`	1.45×10^4	1.53×10^4	1.59×10^4
`fft`	2.05×10^6	2.71×10^6	4.04×10^6
`des`	2.42×10^5	3.66×10^5	6.69×10^5
`fullsearch`	6.25×10^4	9.57×10^5	29.0×10^5
`whetstone`	6.83×10^6	10.2×10^6	14.9×10^6
`dhry`	5.52×10^5	7.53×10^5	13.3×10^5
`matgen`	9.28×10^3	10.9×10^3	17.2×10^3

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 a single CCG is needed. The combined block is called *extended-l-block* or *e-block*.

7 Experimental Results

Our goal is to find a tight bound on a program's WCET. Some 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 microarchitectural modeling which effects the accuracy of the values of $c_{i.j}^{hit}$'s and $c_{i.j}^{miss}$'s in (13).

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 set of benchmark programs from [1] for our evaluation. This set includes 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 lines and the executable code size of the program in 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 measure the program's execution time for this worst case data set. We initialize the program with its assumed worst case data set and then run it in a loop several hundred times and measure the elapsed time. At the beginning of each loop iteration, the instruction cache is flushed. Since this elapsed time includes the overhead to do the loop iteration and cache flushing, we also run an empty loop and measure its execution time. The difference between these two times is the *measured* WCET of the program. We assume that the *measured* WCET of a program is very close to its *actual* WCET.

Table 2 shows the results of our experiments. Clearly the estimated WCET with cache analysis is much tighter than the one without it. For small integer programs (e.g. `check_data` and `piksrt`) it is very close to the measured WCET. The difference between the measured WCET and the estimated WCET is mainly due to the pessimism in the execution-time estimates of function call/return instructions. For other programs, the differences are mainly due to the pessimism in the execution times of floating point instructions.

Table 3 shows, for each program, the number of variables and constraints used, the number of branches in solv-

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	20	4	1	0
`piksrt`	12	0	0	38	22	21	4	1	0
`line`	31	2	264	231	73	60	2	1	0
`circle`	8	1	81	100	24	186	1	1	0
`fft`	31	0	15	92	52	70	12	1	0
`des`	174	11	1,068	550	342	1,165	16+16	5+5	87+86
`fullsearch`	371	3	1,402	678	572	1,754	43	1	28
`whetstone`	52	3	564	400	108	834	13	1	3.5
`dhry`	102	21	607	504	289	794	12+13+13	1+1+1	6+5+5
`matgen`	24	0	0	78	43	42	5	1	0

ing the ILP problem, and the time required to solve the problem. Since each program may have more than one set of functionality constraints [1], 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 solver 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 the solution within the first few calls to the LP solver. The time taken to solve the problem ranges from less than a second to a few minutes on a SGI Indigo2 workstation. With a commercial ILP solver CPLEX, CPU time reduces significantly to a few seconds.

8 Conclusions and Future Work

In this paper, we present a method to find a tight bound on the worst case execution time of embedded software that includes direct-mapped instruction cache analysis. It uses an integer linear programming formulation to solve the problem. This approach does not impose any pessimistic assumptions on the cache activity and it avoids enumeration of program paths. Furthermore, it allows the user to provide additional annotation on feasible and infeasible program paths. The method is implemented in the tool `cinderella`. Experimental results show that the estimated WCET is much closer to the measured WCET than if cache analysis is not included. Since the linear constraints are mostly derived from network flow graphs, the ILP problems are typically solved efficiently.

In future, we would like to extend the current method to handle set-associative instruction caches, as well as data cache memory. We would also like to port `cinderella` to analyze programs running on other hardware platforms.

References

[1] Yau-Tsun Steven Li and Sharad Malik, "Performance analysis of embedded software using implicit path enumeration", in *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, June 1995, pp. 456–461.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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

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

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