Performance Estimation of Embedded Software with Instruction Cache Modeling

YAU-TSUN STEVEN LI, SHARAD MALIK, and ANDREW WOLFE
Princeton University

Embedded systems generally interact in some way with the outside world. This may involve measuring sensors and controlling actuators, communicating with other systems, or interacting with users. These functions impose real-time constraints on system design. Verification of these specifications requires computing an upper bound on the worst-case execution time (WCET) of a hardware/software system. Furthermore, it is critical to derive a tight upper bound on WCET in order to make efficient use of system resources.

The problem of bounding WCET is particularly difficult on modern processors. These processors use cache-based memory systems that vary memory access time based on the dynamic memory access pattern of the program. This must be accurately modeled in order to tightly bound WCET. Several analysis methods have been proposed to bound WCET on processors with instruction caches. Existing approaches either search all possible program paths, an intractable problem, or they use highly pessimistic assumptions to limit the search space. In this paper we present a more effective method for modeling instruction cache activity and computing a tight bound on WCET. The method uses an integer linear programming formulation and does *not* require explicit enumeration of program paths. The method is implemented in the program `cinderella` and we present some experimental results of this implementation.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms: Performance

1. INTRODUCTION

The execution time of a program can often vary significantly from one run to the next on the same system. Even given a known program and a known system, the actual execution time depends on the input data values and the initial state of the system. In many cases it is essential to know the worst-case execution time (WCET) for a hardware/software system. In hard

Authors' address: Department of Electrical Engineering, Princeton University, Princeton, NJ 08544; email: yauli@ee.princeton.edu; sharad@ee.princeton.edu; awolfe@ee.princeton.edu.

real-time systems, the programmer must guarantee that the WCET satisfies the timing deadlines. Many real-time operating systems rely on this 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 large 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.

WCET analysis can be divided into two components: *program path analysis*, which determines the sequence of instructions to be executed in the worst-case scenario, and *microarchitecture modeling*, which models the underlying hardware systems and computes the WCET of a known sequence of instructions. Both components are important in determining tight estimated WCETs.

The program path analysis has been discussed extensively in our previous work [Li and Malik 1995]. It deals with computing the estimated WCET of a program efficiently and making use of the user-provided program path annotations to eliminate infeasible program paths. We observed that while the user annotations do tighten the estimated WCET significantly, a large amount of pessimism still exists in the estimated WCET because of the simple microarchitecture modeling.

Microarchitecture modeling is particularly difficult for modern microprocessors. These processors usually include pipelined instruction execution and cache-based memory systems. These features speed up the typical performance of the system, but complicate timing analysis. The exact execution time of an instruction depends on many factors and varies more than in the previous generation of microprocessors. The cache memory system is particularly difficult to model and is becoming the dominant factor in the pessimism. Incorporating accurate cache modeling into the worst– case timing analysis is essential in order to effectively use modern processors in real-time systems. Our goal is to devise an instruction cache modeling method that accurately represents cache activity and provides enough information for the timing analysis tool to tightly bound the WCET.

We propose a method to model instruction cache memory. Unlike other cache analysis methods, it does not require explicit program path enumeration, yet it provides a tighter bound on the worst-case cache miss penalties than other practical estimation methods that we know of. In this paper, we limit our method to model a direct-mapped instruction cache. However, it can be extended to handle set associative instruction cache memory.

This paper is organized as follows: We first discuss some related work in this area in Section 2. Then, in Section 3, we summarize our previous work in program path analysis. In Section 4 and 5, we describe how this work is extended to model direct-mapped instruction cache. Implementation issues

are discussed in Section 6. This is followed by the experimental results shown in Section 7. Finally, we present the conclusions in Section 8.

2. RELATED WORK

The problem of finding a program's worst-case execution time is in general undecidable and is equivalent to a halting problem. This is true even with a constant-access-time instruction memory. Kligerman and Stoyenko [1986], as well as Puschner and Koza [1989], listed the conditions for this problem to be decidable. These conditions are bounded loops, absence of recursive function calls, and absence of dynamic function calls. These researchers, together with Mok et al. [1989] and Park and Shaw [1992], have proposed a number of methods to determine the estimated WCET. These methods assume a simple hardware model such that the execution time of every instruction in the program is a constant equal to the instruction's worst-case execution time. No cache analysis is performed.

The presence of cache memory complicates the WCET analysis significantly. The reason is that to determine the worst-case execution path, the execution times of individual instructions are needed. Yet without knowing the worst-case execution path, the cache hits and misses of instructions, and hence the execution times of the instructions, cannot be determined. As a result, program path analysis and cache memory analysis are interrelated.

Several WCET analyses with direct-mapped instruction cache modeling methods have recently been proposed. Liu and Lee [1994] noted 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. [1994], who extended Shaw's timing schema methodology [Shaw 1989] 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. Even so, the size of the program for analysis is still limited. Arnold et al. [1994] proposed a less aggressive cache analysis method. They used flow analysis to identify the *potential* cache conflicts and classified each instruction as first miss, always hit, always miss, or first hit categories. This results in fast but less accurate cache analysis. Rawat [1993] handled data cache performance analysis by using graph-coloring techniques. However, this approach had limited success even for small programs. A severe drawback of all the methods above is that they do not accept any user annotations describing infeasible program paths, which are essential in tightening the estimated WCET.

Explicit path enumeration is *not* a necessity in obtaining a 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 show in Section 4, only minimal necessary sequencing information is kept in performing 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) [Park 1992] 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. ILP FORMULATION

Our previous work [Li and Malik 1995] focused on path analysis given a simple microarchitecture model, which assumes that every instruction takes a constant time to execute. Instead of searching all program paths, the path analysis analytically determines the dynamic execution count of each instruction under the worst-case scenario. There are similarities between our analysis technique and the one used by Avrunin et al. [1994] in determining time bounds for concurrent systems.

Since we assume that each instruction takes a constant time to execute, the total execution time can be computed by summing up the product of instruction counts by their corresponding single execution times. Furthermore, since the instructions within a basic block are always executed together, their execution counts must be the same. Hence, they can be considered as a single unit. If we let variable x_i be the execution count of a basic block B_i, and constant c_i be the execution time of the basic block, then the total execution time of the program is given as

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

where N is the number of basic blocks in the program. Clearly, x_is must be integer values. 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 worst-case execution time of a program is transformed into an integer linear programming (ILP) problem which can be solved by many existing ILP solvers.

The linear constraints are divided into two parts: (1) *program structural constraints*, which are derived automatically from the program's control flow graph (CFG), and (2) *mprogram 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 Figure 1, in which a conditional statement is nested inside a while loop. Figure 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 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 extracted from this example are

$$d_1 = 1 \tag{2}$$

$$x_1 = d_1 = d_2 \tag{3}$$

$$x_2 = d_2 + d_8 = d_3 + d_9 \tag{4}$$

$$x_3 = d_3 = d_4 + d_5 \tag{5}$$

$$x_4 = d_4 = d_6 \tag{6}$$

$$x_5 = d_5 = d_7 \tag{7}$$

$$x_6 = d_6 + d_7 = d_8 \tag{8}$$

$$x_7 = d_9 = d_{10}. \tag{9}$$

Here, the first constraint (2) specifies that the code fragment is to be executed once. The structural constraints do not provide any loop bound information. This information is provided by the user by using functionality constraints. In this example, we note that since variable `k` is positive before the program control 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

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

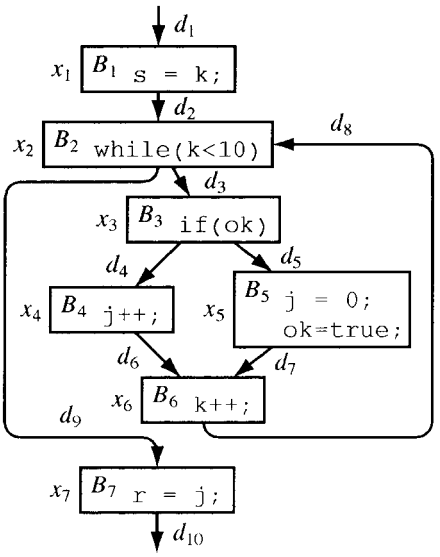
The functionality constraints can also be used to specify other path information. For example, we observe that the `else` statement (B_5) can be executed at most once inside the loop. This information can be specified as

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

All of these constraints— (2) through (11)— are passed to the ILP solver with the goal of maximizing the cost function (1). The ILP solver returns a bound on the worst-case execution time and the execution counts of the basic blocks.

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

(i) Source code



(ii) Control flow graph (CFG)

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

4. DIRECT-MAPPED INSTRUCTION CACHE ANALYSIS

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

4.1 Modified Cost Function

With cache memory, each instruction fetch will result in either a cache hit or a cache miss, which may in turn result in two very different instruction execution times. The simple microarchitecture model that each instruction takes a constant time to execute no longer models this situation accurately. 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*. An *l-block* is defined as a contiguous sequence of code within the same basic block that is mapped to the same cache set in the instruction cache. In other words, the *l-blocks* are formed by the intersection of basic blocks with the cache set line size. All instructions within a *l-block* are always executed together in sequence. Further, since the cache controller always loads a line of code into the cache, these instructions are either

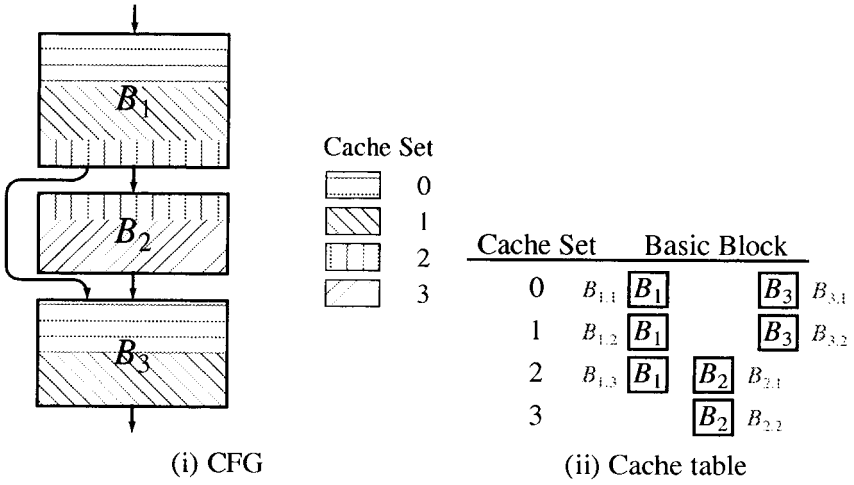


Fig. 2. An example showing how the *l-blocks* are constructed. Each rectangle in the cache table represents a *l-block*.

in the cache completely or not in the cache at all. These are denoted as a cache hit or a cache miss, respectively, of the *l-block*.

Figure 2(i) shows a CFG with 3 basic blocks. Suppose that the instruction cache has 4 cache sets. Since the starting address of each basic block can be determined from the program's executable code, we can find all cache sets that each basic block is mapped to, and add an entry on these cache lines in the cache table (Figure 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* $B_{i.1}$, $B_{i.2}$, . . . , $B_{i.n_i}$.

For any two *l-blocks* that are mapped to the same cache set, they will *conflict* with each other if they have different address tags. The execution of one *l-block* will displace the cache content of the other. For instance, *l-block* $B_{1.1}$ conflicts with *l-block* $B_{3.1}$ in Fig.ure 2. There are also cases where two *l-blocks* do not conflict with each other. This situation happens when the basic block boundary is not aligned with the cache line boundary. For instance, *l-blocks* $B_{1.3}$ and $B_{2.1}$ in Figure 2, each occupies a partial cache line and they do not conflict with each other. They are called *nonconflicting l-blocks*.

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 $x_{i.j}^{hit}$ and $x_{i.j}^{miss}$,respectively, and

$$x_i = x_{i.j}^{hit} + x_{i.j}^{miss}, \quad j = 1,2, \ldots, n_i. \tag{12}$$

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

$$\text{Total execution time} = \sum_{i=1}^{N} \sum_{j=1}^{n_i} (c_{i,j}^{hit} x_{i,j}^{hit} + c_{i,j}^{miss} x_{i,j}^{miss}) \tag{13}$$

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

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* $B_{k.l}$ mapped 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,

$$x_{k.l}^{miss} \leq 1. \tag{14}$$

A slightly more complicated case occurs when two or more *nonconflicting l-blocks* are mapped to the same cache line, such as $B_{1.3}$ and $B_{2.1}$ in Fig.ure 2. Since the cache controller always fetches a line of code into the cache, the execution of any of the *l-blocks* will result in the cache controller loading all of them into the cache line. Therefore, the sum of their cache miss counts is at most one. In this example, the constraint is

$$x_{1.3}^{miss} + x_{2.1}^{miss} \leq 1. \tag{15}$$

When a cache line contains two or more *conflicting l-blocks*, the hit/miss counts of these *l-blocks* will be effected by their execution sequence. An important observation is that the execution of any other *l-blocks* mapped to other cache sets will have no effect on these counts. This leads us to examine the control flow of *l-blocks* mapped to a particular cache set by using a *cache conflict graph*.

4.3 Cache Conflict Graph

A cache conflict graph (CCG) is constructed for every cache set 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 cache set. 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 set. 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

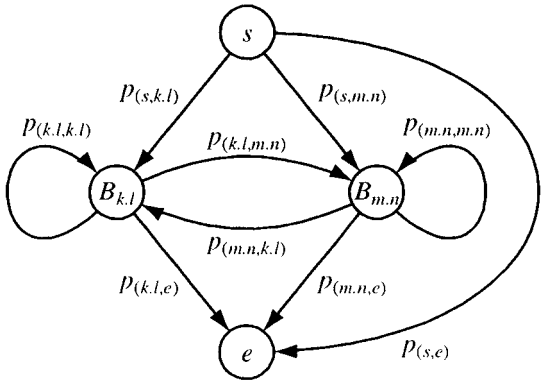


Fig. 3. A general cache conflict graph containing two conflicting *l-blocks*.

l-blocks of the same cache set, 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 Figure 3. The program control begins at the start node. After executing some other *l-blocks* from other cache lines, it will eventually reach node $B_{k.l}$, node $B_{m.n}$, or the end node. Similarly, after executing $B_{k.l}$, the program control may pass through some *l-blocks* from other cache lines and then reach 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 program 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 $B_{i.j}$:

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

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:

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

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. For a direct mapped cache, each cache set has one cache line. 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 the 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,

$$p_{(i.j,\ i.j)} \leq x_{i.j}^{hit} \leq p_{(s,\ i.j)} + p_{(i.j,\ i.j)}. \tag{18}$$

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

$$x_{i.j}^{hit} = p_{(i.j,\ i.j)}. \tag{19}$$

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 is 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 set.

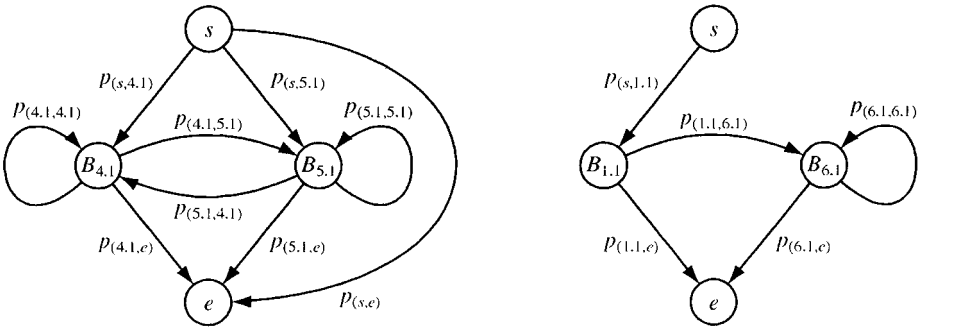
The above constraints can also be used to solve the best-case execution time. The objective is to minimize cost function (13), subject to the same structural constraints, functionality constraints, and cache constraints. In this case the ILP solver will try to increase the value of $x_{i.j}^{hit}$ as much as possible. If $p_{(i.j,\ i.j)}$ (self-edge variable) exists, then the ILP solver may set $p_{(i.j,\ i.j)} = x_{i.j}^{hit} = x_i$. However, this is not possible in any execution trace. Before this path can occur, the program 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:

$$x_i \leq Z \sum_{u.v,\ u.v \neq i.j} p_{(u.v,\ i.j)}, \tag{20}$$

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

4.4 A Simple Example

Section 3 shows how the structural constraints and functionality constraints are constructed for the example shown in Figure 1. For simplicity, assume that each basic block will only be partitioned into one *l-block*. Consider the cases where the `if` statement ($B_{4.1}$) and `else` statement ($B_{5.1}$) conflict with each other and the loop preheader ($B_{1.1}$) conflicts with the loop body statement ($B_{6.1}$).



(i) *l-block* $B_{4.1}$ conflicts with *l-block* $B_{5.1}$ (ii) *l-block* $B_{1.1}$ conflicts with *l-block* $B_{6.1}$

Fig. 4. The two CCGs of the example shown in Figure 1.

Figure 4 shows the CCGs for these two cases. For *l-blocks*, $B_{4.1}$ and $B_{5.1}$ (Figure 4(i)), the constraints for the worst-case execution estimation are

$$x_4 = x_{4.1}^{hit} + x_{4.1}^{miss} \tag{21}$$

$$x_5 = x_{5.1}^{hit} + x_{5.1}^{miss} \tag{22}$$

$$x_4 = p_{(s,\,4.1)} + p_{(4.1,4.1)} + p_{(5.1,4.1)} = p_{(4.1,\,e)} + p_{(4.1,4.1)} + p_{(4.1,5.1)} \tag{23}$$

$$x_5 = p_{(s,\,5.1)} + p_{(5.1,5.1)} + p_{(4.1,5.1)} = p_{(5.1,\,e)} + p_{(5.1,5.1)} + p_{(5.1,4.1)} \tag{24}$$

$$p_{(s,\,4.1)} + p_{(s,\,5.1)} + p_{(s,\,e)} = 1 \tag{25}$$

$$p_{(4.1,4.1)} \leq x_{4.1}^{hit} \leq p_{(s,\,4.1)} + p_{(4.1,4.1)} \tag{26}$$

$$p_{(5.1,5.1)} \leq x_{5.1}^{hit} \leq p_{(s,\,5.1)} + p_{(5.1,5.1)}. \tag{27}$$

Some further path information can be provided here by the user. We note that if the `if` statement is executed, it implies that variable `ok` is true, and therefore the `else` statement will never be executed again. So there will never be a control flow from basic block B_4 to basic block B_5. This information can be expressed as

$$p_{(4.1,5.1)} = 0. \tag{28}$$

The cache constraints for the second case are

$$x_1 = x_{1.1}^{hit} + x_{1.1}^{miss} \tag{29}$$

$$x_6 = x_{6.1}^{hit} + x_{6.1}^{miss} \tag{30}$$

$$x_1 = p_{(s,\,1.1)} = p_{(1.1,6.1)} + p_{(1.1,\,e)} \tag{31}$$

$$x_6 = p_{(1.1,6.1)} + p_{(6.1,6.1)} = p_{(6.1,\,e)} + p_{(6.1,6.1)} \tag{32}$$

$$p_{(s,\,1.1)} = 1 \tag{33}$$

$$x_{1.1}^{hit} \leq p_{(s,\,1.1)} \tag{34}$$

$$x_{6.1}^{hit} = p_{(6.1,6.1)}. \tag{35}$$

Since all other *l-blocks* are nonconflicting *l-blocks*, their cache constraints are

$$x_2 = x_2^{hit} + x_2^{miss} \tag{36}$$

$$x_3 = x_3^{hit} + x_3^{miss} \tag{37}$$

$$x_7 = x_7^{hit} + x_7^{miss} \tag{38}$$

$$x_2^{miss} \leq 1 \tag{39}$$

$$x_3^{miss} \leq 1 \tag{40}$$

$$x_7^{miss} \leq 1. \tag{41}$$

4.4.1 *Bounds on p-Variables*. In this section, we discuss bounds on the *p*-variables. Without the correct bounds, the ILP solver may return an infeasible *l-block* count and an overly pessimistic estimated WCET. This is demonstrated by the example in Figure 5. 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 (Figure 5(ii)) and the following cache constraints generated:

$$x_4 = p_{(s,\,4.1)} + p_{(4.1,4.1)} + p_{(7.1,4.1)} = p_{(4.1,\,e)} + p_{(4.1,4.1)} + p_{(4.1,7.1)} \tag{42}$$

$$x_7 = p_{(s,\,7.1)} + p_{(7.1,7.1)} + p_{(4.1,7.1)} = p_{(7.1,\,e)} + p_{(7.1,7.1)} + p_{(7.1,4.1)} \tag{43}$$

$$p_{(s,\,4.1)} + p_{(s,\,7.1)} + p_{(s,\,e)} = 1 \tag{44}$$

$$p_{(4.1,4.1)} \leq x_{4.1}^{hit} \leq p_{(s,\,4.1)} + p_{(4.1,4.1)} \tag{45}$$

$$p_{(7.1,7.1)} \leq x_{7.1}^{hit} \leq p_{(s,\,7.1)} + p_{(7.1,7.1)}. \tag{46}$$

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

$$x_3 = 10x_1, \quad x_7 = 10x_5, \quad x_4 = 9x_1. \tag{47-49}$$

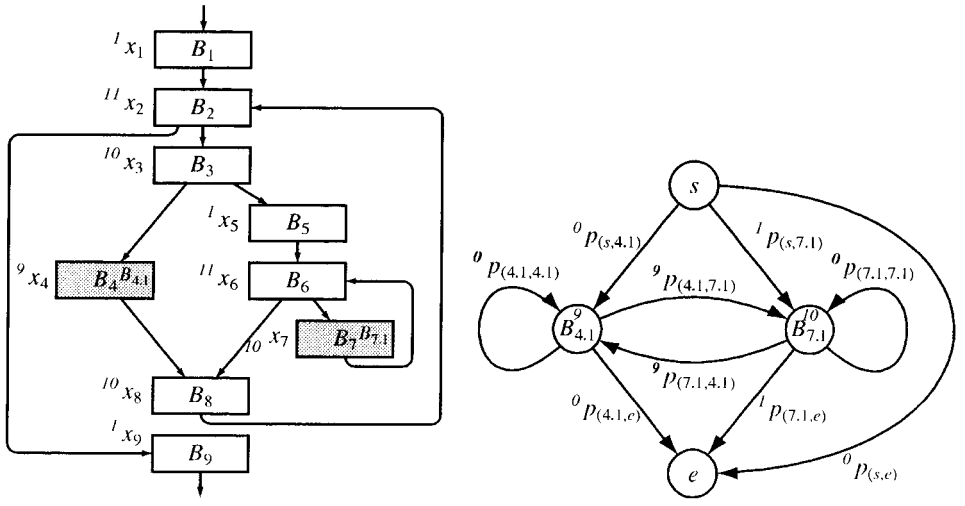


Fig. 5. Example showing two conflicting *l-blocks* ($B_{4.1}$ and $B_{7.1}$) from two different loops. Italicized numbers on the left of the variables are the pessimistic worst-case solution returned from ILP solver.

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 shown on the upper left corner 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* $B_{7.1}$ executed first. This execution sequence generates 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 is entered only once. Once the program control enters the inner loop, *l-block* $B_{7.1}$ must be executed 10 times before the program control exits the inner loop. Hence, there must be at least 9 cache hits for *l-block* $B_{7.1}$. The ILP solver overestimates 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. The flow equations (16) generated from the CCG implicitly bound the p-variables as follows: For any variable $p_{(i.j,\ u.v)}$, its bounds are

$$0 \leq p_{(i.j,\ u.v)} \leq \min(x_i, x_u). \tag{50}$$

Consider the case where 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 is 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) is 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

$$p_{(i.j,\ u.v)} \leq x_h \tag{51}$$

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

In general, a constraint is constructed at each loop preheader. All paths going 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:

$$p_{(s,\ 7.1)} + p_{(4.1,7.1)} \leq x_5. \tag{52}$$

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 cache analysis is performed when there are function calls in the program.

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, respectively, of the *l-block* $B_{i.j}$. Equation (12) is still valid and (13) still represents the total execution time of the program.

In performing cache analysis, we need to consider the cache conflicts among *l-blocks* from different functions and the bounds of the p-variables. For these reasons, every function call is treated as if it were inlined. During the construction of CFG, when a function call occurs, an f-edge that contains an instance of the callee function's CFG is used. The edge has a variable f_k that represents the number of times that a particular instance of the callee function is executed. Each variable and name in the callee

[1]A loop preheader is the basic block just before entering the loop. For instance, in the example shown in Figure 5, 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.

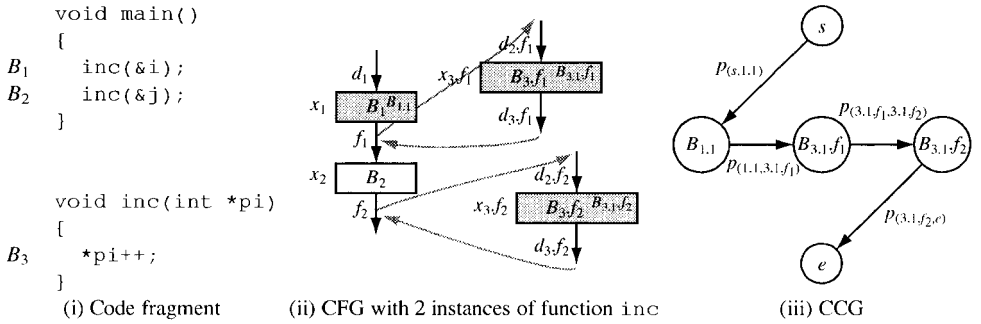


Fig. 6. An example code fragment showing how function calls are handled.

function has a suffix '$.f_k$' to distinguish it from other instances of the same callee function.

Consider the example in Figure 6. Here, function `inc` is called twice in the `main` function. The CFG is shown in Figure 6(ii), where two instances of function `inc`'s CFG are created. The structural constraints are

$$d_1 = 1 \tag{53}$$

$$x_1 = d_1 = f_1 \tag{54}$$

$$x_2 = f_1 = f_2 \tag{55}$$

$$d_2.f_1 = f_1 \tag{56}$$

$$x_3.f_1 = d_2.f_1 = d_3.f_1 \tag{57}$$

$$d_2.f_2 = f_2 \tag{58}$$

$$x_3.f_2 = d_2.f_2 = d_3.f_2 \tag{59}$$

$$x_3 = x_3.f_1 + x_3.f_2. \tag{60}$$

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 path information among different functions. The user can also provide path information on any instance of the function.

The CCG is constructed as before, by treating each instance of the *l-block* $B_{i.j}.f_k$ as independent from other instances of the same *l-block*. In the example shown in Figure 6, if *l-block* $B_{1.1}$ conflicts with *l-block* $B_{3.1}$, then, since *l-block* $B_{3.1}$ has 2 instances ($B_{3.1}.f_1$ and $B_{3.1}.f_2$), there will be 5 nodes in the CCG (Figure 6(iii)).

The cache constraints and the bounds on p variables are constructed as before, except that 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* ($B_{i,j}.f_l$) represents cache hits 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 cache constraints derived from the example's CCG are

$$x_1 = x_{1.1}^{hit} + x_{1.1}^{miss} \tag{61}$$

$$x_2 = x_{2.1}^{hit} + x_{2.1}^{miss} \tag{62}$$

$$x_3 = x_{3.1}^{hit} + x_{3.1}^{miss} \tag{63}$$

$$x_{2.1}^{miss} \leq 1 \tag{64}$$

$$x_1 = p_{(s,\,1.1)} = p_{(1.1,3.1.f_1)} \tag{65}$$

$$x_3.f_1 = p_{(1.1,3.1.f_1)} = p_{(3.1.f_1,\,3.1.f_2)} \tag{66}$$

$$x_3.f_2 = p_{(3.1.f_1,\,3.1.f_2)} = p_{(3.1.f_2,\,e)} \tag{67}$$

$$x_{1.1}^{hit} = 0 \tag{68}$$

$$x_{3.1}^{hit} = p_{(3.1.f_1,\,3.1.f_2)} \tag{69}$$

$$p_{(s,\,1.1)} = 1. \tag{70}$$

6. IMPLEMENTATION AND HARDWARE MODELING

The above cache analysis method has been integrated into our original tool `cinderella`, which estimates the WCET of programs running on an Intel QT960 development board [Intel Corporation 1990] containing a 20MHz Intel i960KB processor, 128KB of main memory, and several I/O peripherals. The i960KB processor is a 32-bit RISC processor used in many embedded systems (e.g., in laser printers). The processor 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 for faster execution of function call/return instructions [Intel Corporation 1991; Myers and Budde 1988].

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 pessmism in the final estimated WCET. Additional time is also added to the last *l-block* of each basic block to ensure that all buffered load/store instructions [Intel Corporation 1991] are completed when the program control reaches the end of the basic block. Note that the instruction boundary may not be aligned with the cache line boundary, i.e., an

instruction may span two cache sets, and consequently may span two *l-blocks*. When this happens, the effective execution time of this instruction is counted in the hit cost of the second *l-block*. Since these two *l-blocks* must come from the same basic block, they have the same total execution count. Therefore, the total execution time spent on this instruction is correctly accounted for. The miss cost $c_{i,j}^{miss}$ of the *l-block* is equal to the hit cost $c_{i,j}^{hit}$, plus the time needed to load the line of code into the cache memory.

`Cinderella`[2] now contains about 20,000 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 xs and fs are labeled along with the program's source code. The user is then asked to provide loop bounds. A WCET bound can be computed at this point. The user can provide additional path information, if available, to tighten this bound. We use a public domain ILP solver `lp_solve`[3] 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 adjacent cache lines hold instructions from the same set of basic blocks, e.g., cache lines 0 and 1 in Figure 2(ii), then the corresponding *l-blocks* can be combined together and only one CCG is drawn for those cache lines.

7. EXPERIMENTAL RESULTS

In this section,we evaluate the accuracy and performance of our cache analysis method. Since there are no standard benchmark programs, we selected a set of benchmark programs from a variety of sources. They included programs from academic sources, DSP routines, standard software benchmarks, and a JPEG decompression program, which is the largest and most complicated one used in this kind of analysis. Table I shows for each program, its name, description, source code line size, and i960KB binary code size in bytes.

7.1 Measurement Methods

Ideally, we would like to compare each program's estimated WCET with its actual WCET. But since it is impractical to simulate every possible input data set and every initial system state, a program's actual WCET may not be determined. In fact, if we can determine a program's actual WCET, then we do not need its estimated WCET anymore!

To resolve this problem, we tried to identify each program's *worst-case input data set* and compared its execution time, denoted *measured WCET*,

[2]Details of this tool can be obtained via `http://www.ee.princeton.edu/nyauli/cinderella-3.0/`.

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

Table I. Benchmark Examples: Descriptions, Source File Line and i960KB Binary Code Sizes

Program	Description	Lines	Bytes
check data	Check if any of the elements in an array is negative, from Park [1992]	23	88
circle	Circle drawing routine, from Gupta [1993]	100	1,588
des	Data Encryption Standard	192	1,852
dhry	Dhrystone benchmark	761	1,360
djpeg	Decompression of 128 × 96 color JPEG image	857	5,408
fdct	JPEG forward discrete cosine transform	300	996
fft	1024-point Fast Fourier Transform	57	500
line	Line drawing routine, from Gupta [1993]	165	1,556
matcnt	Summation of 2 100 × 100 matrices, from Arnold [Arnold, Mueller, Whalley, and Harmon 1994]	85	460
matcnt2	Matcnt with inlined functions	73	400
piksrt	Insertion sort of 10 elements	19	104
sort	Bubble sort of 500 elements, from Arnold [Arnold, Mueller, Whalley, and Harmon 1994]	41	152
sort2	Sort with inlined functions	30	148
stats	Calculate the sum, mean and variance of two 1,000-element arrays, from Arnold [Arnold, Mueller, Whalley, and Harmon 1994]	100	656
stats2	Stats with inlined functions	90	596
whetstone	Whetstone benchmark	196	2,760

with the estimated WCET. We assumed that the worst-case initial system state was the one with empty cache contents, since this generates the maximum number of cache misses. For most programs, the worst-case input data set could be determined. For instance, the worst-case input data set for the sorting programs is an array of descending elements. For these programs, the measured WCET should be very close to the actual WCET. But for more complicated programs, this became a nontrivial task. These programs have instructions whose execution times are data-dependent, such as the floating point instructions presented in programs `fft` and `fdct`, and/or complicated input data-dependent execution paths, such as programs `des` and `djpeg`. For these four programs, the worst-case data input sets are unknown. We generated a series of random input data sets, measured their corresponding execution times, and picked the longest one as the measured WCET of the program. In this case, the difference between the measured WCET and the actual WCET may be larger. To determine the *measured WCET* accurately, we executed each program on the Intel QT960 evaluation board and used a logic analyzer to measure its execution time. The difference between a program's estimated WCET and its measured WCET was equal to the difference between the estimated WCET and the actual WCET, due to pessimism in path analysis, cache modeling, and execution unit modeling, plus the difference between the actual WCET and the measured WCET.

Table II. Estimated WCETs of Benchmark Programs. Estimated WCETs and Measured WCETs In Units of Clock Cycles

Program	Measured WCET	Estimated WCET	Ratio
check data	4.30×10^2	4.91×10^2	1.14
circle	1.45×10^4	1.54×10^4	1.06
des	2.44×10^5	3.70×10^5	1.52
dhry	5.76×10^5	7.57×10^5	1.31
djpeg	3.56×10^7	7.04×10^7	1.98
fdct	9.05×10^3	9.11×10^3	1.01
fft	2.20×10^6	2.63×10^6	1.20
line	4.84×10^3	6.09×10^3	1.26
matcnt	2.20×10^6	5.46×10^6	2.48
matcnt2	1.86×10^6	2.11×10^6	1.13
piksrt	1.71×10^3	1.74×10^3	1.02
sort	9.99×10^6	27.8×10^6	2.78
sort2	6.75×10^6	7.09×10^6	1.05
stats	1.16×10^6	2.21×10^6	1.91
stats2	1.06×10^6	1.24×10^6	1.17
whetstone	6.94×10^6	10.5×10^6	1.51

7.2 Results

Table II shows the results of our experiments. The second and third columns show, respectively, the measured WCET and estimated WCET with cache analysis . The fourth column shows the ratio of the estimated WCET to the measured WCET. All ratios are larger than one, meaning that all estimated WCETs bound their corresponding measured WCETs. Smaller ratios mean tighter estimates.

For small programs like `check data` and `piksrt`, the estimated WCETs are very close to their corresponding measured WCETs. Programs `sort`, `matcnt`, and `stats` have larger than expected pessimism. This is because we did not model the register windows featured in the i960KB processor. We conservatively assumed that the register window overflowed (underflowed) on each function call (return). This pessimism incurred about 50 clock cycles on each function call and function return. Since the above programs had lots of small function calls, a large amount of pessimism resulted. In order to factor out this pessimism, we inlined the frequently called functions in these programs. The modified programs are `sort2`, `matcnt2`, and `stats2`. Their estimated WCETs are much tighter than the original ones. We are currently working on this problem to reduce the pessimism.

The pessimism for programs with floating point instructions, such as `fft` and `whetstone`, is also higher. The reason is that the execution time of an i960KB floating point instruction is data-dependent. For our worst-case estimation, we conservatively assumed that each floating point instruction took its worst-case execution time to complete, which is typically 30%–40% more than its average execution time [Intel Corporation 1991].

Finally, the reason for large pessimism in program `djpeg` is due to the loose measured WCET. For the worst-case estimation, we assumed that the

input image was so random that no compression was achieved in the Huffman encoding process. Therefore, the Huffman decoding function in `djpeg` needs to loop more in reading and decoding the Huffman symbols. But for all random images we generated for determining the measured WCET, some sort of compression was still achieved. The larger differences between the loop bounds in the estimation and the actual loop iterations in measurements accounted for this pessimism. In this program, the measured WCET might not be as close to its actual WCET as other programs are. This illustrates that the actual WCET is in some cases very hard to be attained through simulation, whereas static analysis always guarantees bounding the actual WCET.

A few simple programs were also used by Arnold et al. [1994] and Lim et al. [1994]. It is natural to compare our estimated WCETs to theirs. However, the above researchers used different hardware platforms for modeling. Arnold et al. used a Sparc simulator and Lim et al. used a MIPS R3000 evaluation board. Since the binary codes were different and the model of each processor was different, there is no direct way to compare the results. One main drawback of the above researchers' methods is that they cannot accept path information other than loop bounds. Therefore, for programs like `sort` and `fft`, which have nested loops in which the loop iteration of the inner loop depends on the loop index of the outer loop, they reported estimated WCETs that are roughly two times the measured WCETs. Our analysis method is superior in that it accepts path information even in the presence of instruction cache analysis. This results in much tighter estimation, and our method can analyze more complicated programs.

Since a large amount of pessimism shown in Table II is due to the pessimism in modeling the execution unit, we would like to factor it out. For programs whose code size is greater than 512-bytes, we executed each program with its worst-case input data set to generate the instruction address trace. This was passed to a cache simulator `DineroIII` to determine the number of cache misses. We then used `cinderella` to estimate the program's worst-case cache misses by using a hardware model in which a cache miss incurs one unit of time and a cache hit and other execution times incur zero execution times. Table III shows the results. For many programs, the simulator and `cinderella` reported the same number of cache misses. The pessimism in other programs is mainly due to inaccuracy in path analysis. In particular, program `djpeg` has exceptionally large pessimism because the Huffman decoding function contains conflicted code. Since the execution count of this function is conservatively overestimated, a large number of cache misses resulted.

7.3 Performance Issues

The structural and cache constraints are derived from the CFG and CCGs that are very similar to network flow graphs. We therefore expect that the ILP solver can solve the problem efficiently. Table IV shows, for each

Table III. Estimated Worst-Case Number of Cache Misses of Benchmark Programs. The instruction cache is 512–byte direct-mapped, its line size is 16 bytes

Program	DineroIII simulation	Est. worst case cache misses	Ratio
circle	443	458	1.03
des	3,872	4,188	1.08
dhry	8,304	8,304	1.00
djpeg	230,861	316,394	1.37
fdct	63	63	1.00
line	99	101	1.02
stats	47	47	1.00
stats2	44	44	1.00
whetstone	18,678	18,678	1.00

Table IV. Performance Issues in Cache Analysis

	No. of Variables				No. of Constraints			ILP	Time
Program	*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
circle	8	1	81	100	24	186	1	1	0
des	174	11	728	560	342	1,059	16+16	13+13	171+197
dhry	102	21	503	504	289	777	24×4+26×4	1×8	0×3+2+0 +1×2+4
djpeg	296	20	1,816	416	613	2,568	64	1	87
fdct	8	0	18	34	16	49	2	1	0
fft	27	0	0	80	46	46	11	1	0
line	31	2	264	231	73	450	2	1	3
matcnt	20	4	0	106	59	61	4	1	0
matcnt2	20	2	0	92	49	54	4	1	0
piksrt	12	0	0	42	22	26	4	1	0
sort	15	1	0	58	35	31	6	1	0
sort2	15	0	0	50	30	27	6	1	0
stats	28	13	75	180	99	203	4	1	0
stats2	28	7	41	144	75	158	4	1	0
whetstone	52	3	301	388	108	739	14	1	2

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 [Li and Malik 1995], 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 is called n times. The + symbol is used again 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 could still find an integer solution within the first few calls to the linear programming solver. The time taken to solve the problem ranged from less than a second to a few minutes on a SGI Indigo2

Table V. Complexity of the ILP Problem: Number of Cache Lines Doubled to 64

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

workstation. With a commercial ILP solver, CPLEX, the CPU time reduced significantly to a few seconds.

In order to evaluate how the cache size affects solving time, we doubled the number of cache lines (and hence the cache size) from 32 lines to 64 lines and determined the CPU time needed to solve the ILP problems. Table V shows the results. From the table, we determined that the number of variables and constraints changed little when the number of cache lines is doubled. The time to solve the ILP problem is of the same order as before. The primary reason is that although increasing the number of cache lines increases 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 each other out.

8. CONCLUSIONS AND FUTURE WORK

In this paper we presented a method to determine a tight bound on the worst-case execution time of a given program. 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 additional program path information so that a tighter bound may be obtained. The method is implemented in `cinderella`, and the experimental results show that the estimated WCETs tightly bound the corresponding measured WCETs. Since the linear constraints are mostly derived from the network flow graphs, the ILP problems are typically solved efficiently.

We extended this method to analyze a set-associative instruction cache. `Cinderella` has been ported to model the Motorola M68000 architecture. We are now working on data cache modeling, as well as refining our microarchitecture modeling to model register windows and other advanced microarchitecture features.

ACKNOWLEDGMENTS

This research work was supported by a grant from ONR (grant N00014-95-0274). We gratefully acknowledge numerous useful suggestions from Wayne Wolf, Margaret Martonosi, and Janak Patel. We also thank Rajesh Gupta and David Whalley for providing some of the benchmark programs.

REFERENCES

ARNOLD, R., MUELLER, F., WHALLEY, D., AND HARMON, M. 1994. Bounding worst-case instruction cache performance. In *Proceedings of the 15th IEEE Symposium on Real-Time Systems* (Dec.). IEEE Computer Society Press, Los Alamitos, CA, 172–181.

AVRUNIN, G. S., CORBETT, J. C., DILLON, L. K., AND WILEDEN, J. C. 1994. Automated derivation of time bounds in uniprocessor concurrent systems. *IEEE Trans. Softw. Eng. 20*, 9 (Sept. 1994), 708–719.

GUPTA, R. K. 1994. Co-synthesis of hardware and software for digital embedded systems. Ph.D. Dissertation. Stanford University, Stanford, CA.

INTEL CORPORATION, 1990. QT960 User Manual. Intel Corp., Santa Clara, CA.

INTEL CORPORATION, 1991. i960KA/KB Microprocessor Programmers's Reference Manual. Intel Corp., Santa Clara, CA.

KLIGERMAN, E AND STOYENKO, A D 1986. Real-time Euclid: a language for reliable real-time systems. *IEEE Trans. Softw. Eng. SE-12*, 9 (Sept. 1986), 941–949.

LI, Y.-T. S. AND MALIK, S. 1995. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation* (DAC '95, San Francisco, CA, June 12–16, 1995), B. T. Preas, Ed. ACM Press, New York, NY, 456–461.

LIM, S., BAE, Y. H., JANG, G. T., RHEE, B., MIN, S. L., PARK, C. Y., SHIN, H., PARK, K., AND KIM, C. S. 1994. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of the 15th IEEE Symposium on Real-Time Systems* (Dec.). IEEE Computer Society Press, Los Alamitos, CA, 97–108.

LIU, J. AND LEE, H. 1994. Deterministic upperbounds of the worst-case execution times of cached programs. In *Proceedings of the 15th IEEE Symposium on Real-Time Systems* (Dec.). IEEE Computer Society Press, Los Alamitos, CA, 182–191.

MOK, A. K., AMERASINGHE, P., CHEN, M., AND TANTISIRIVAT, K. 1989. Evaluating tight execution time bounds of programs by annotations. In *Proceedings of the Sixth IEEE Workshop on Real-Time Operating Systems and Software* (May). IEEE Computer Society Press, Los Alamitos, CA, 74–80.

MYERS, G. J. AND BUDDE, D. L. 1988. *The 80960 Microprocessor Architecture*. John Wiley & Sons, Inc., New York, NY.

PARK, C. Y. 1992. Predicting deterministic execution times of real-time programs. Ph.D. Dissertation. University of Washington, Seattle, WA.

PUSCHNER, P. AND KOZA, CH. 1989. Calculating the maximum, execution time of real-time programs. *Real-Time Syst. 1*, 2 (Sep. 1989), 159–176.

RAWAT, J. 1993. Static analysis of cache performance for real-time programming. Master's Thesis. Iowa State Univ., Ames, IA.

SHAW, A. C. 1989. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng. 15*, 7 (July 1989), 875–889.

Received: October 1995; revised: September 1996; accepted: December 1997