Cache Modeling for Real-Time Software:
Beyond Direct Mapped Instruction Caches

Yau-Tsun Steven Li        Sharad Malik        Andrew Wolfe
Department of Electrical Engineering, Princeton University,
Princeton, NJ 08544, USA.
{yauli,sharad,awolfe}@ee.princeton.edu

Abstract

This paper presents a method for determining a tight bound on the worst case execution time of a program when running on a given hardware system with cache memory. Caches are used to improve the average memory performance — however, their presence complicates the worst case timing analysis. Any pessimistic predictions on cache hits/misses will result in loose estimation. In our previous research in this area, we built an Integer-Linear-Programming solution for this problem which included analysis of direct mapped instruction caches. In this paper, we describe the complex extensions of this technique to deal with set associative instruction caches, data caches and unified caches. We believe that this research now provides a comprehensive solution to the problem of worst-case performance analysis of software running on processors with caches. These techniques have been implemented in a design tool Cinderella. Some experimental results are presented that demonstrate the practical applicability of this analysis.

1. Introduction

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

The actual WCET of a program cannot be determined unless we simulate all possible combinations of input data values and initial system states. This is clearly impractical due to the exponentially large number of simulations required. As a result, we often obtain an estimate of 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.

Our objective in this paper is to determine the estimated WCET of a given program when running on a hardware system with cache memory. Caches are present in many hardware systems to improve the overall memory performance. Many modern processors have on-chip caches. The drawback of caches is that it complicates the timing analysis. Although one can conservatively assume that every memory access result in a cache miss in determining the estimated WCET, this assumption yields a very loose estimation for it to be practically useful.

Thus, we see that an efficient and accurate cache analysis method is essential in analyzing real-time systems. However, we would also like to stress that this is only one of the prerequisites for determining tight estimated WCETs. Information regarding infeasible program paths, and accurate pipeline analysis are equally important in determining tight estimated WCETs. Thus, a good cache analysis method needs to integrate these capabilities while at the same time be capable of handling different kinds of cache organizations.

2. Related Work

Several research groups [2, 7, 8] have proposed different WCET analysis methods with direct mapped instruction cache modeling. The main drawback of the above approaches is that the number of iterations of all loops in the program is assumed to be fixed for each loop entry, and it is not possible to utilize any additional functional information regarding the iteration counts. This results in loose estimation for programs with irregular loop bounds or with many conditional statements. In our earlier work, we presented direct mapped instruction cache modeling based on an integer linear programming (ILP) formulation [6]. This method overcomes the above shortcoming. It can analyze larger, more complicated programs and return tighter estimates.

Data cache analysis is more difficult than instruction cache analysis because the data address may change when the same load/store instruction is executed repeatedly. In many cases, the address may be ambiguous and even im-
possible to determine. In Hur et al.'s work [4], all data memory accesses with unknown addresses are treated as two cache misses. This overly pessimistic method generates results that are even worse than the ones without data cache analysis. This is later improved in the work by Kim et al. [5], who use linear Diophantine equations to analyze nested loops. The limitation of this method is that the memory accessed within the loop body must be fitted entirely into the data cache. Rawat [9] used graph-coloring techniques to analyze data cache performance. This approach has limited success even for small programs.

In this paper we handle the complex extensions of our previous work to handle set associative caches, data caches and unified caches. We will briefly describe the direct mapped instruction cache modeling in the following section and then show how it is extended to model other caches in Sections 4 and 5.

3. ILP Formulation

Our previous work [6] in timing analysis is based on an integer linear programming formulation. We partition the problem of determining a program’s WCET into two subproblems: program path analysis and microarchitecture modeling. Program path analysis handles the program path annotations that describe feasible and infeasible paths, and determines the set of paths that correspond to the program’s WCET. Microarchitecture modeling models the underlying hardware system, which may contain the CPU pipeline and direct mapped instruction cache.

The problem of determining a program’s WCET is transformed into a set of ILP problems. We briefly describe this method in the following subsections.

3.1. Program Path Analysis

The focus of this analysis is on how to formulate the linear constraints so that every feasible program path is included in the solution space bounded by these constraints. We also show how to use the linear constraints to exclude the infeasible program paths.

Here, we assume a simple microarchitecture such that each basic block $B_i$ of the program takes a constant time $c_i$ to execute. Let variable $x_i$ be the execution count of the basic block $B_i$ and there be $N$ basic blocks in the program. Then the total execution time of the program is given by the linear expression:

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

Clearly, the $x_i$'s must be integer values. They are constrained by the program's structure, as well as its data val-

---

1A basic block is a maximum sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

...ues which may alter the flow of the program. These constraints are modeled by program structural constraints and program functionality constraints respectively. The former constraints are derived automatically from the program’s control flow graph (CFG) [1], and the latter can be derived by performing data flow analysis, or more effectively provided by the user.

The construction of these constraints is best illustrated by an example shown in Fig. 1, in which a conditional statement is nested inside a while loop. Fig. 1(ii) shows the CFG. Each node in the CFG represents a basic block. 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, at each node, the basic block execution count is equal to the number of times that the control enters the node (inflow), and is also equal to the number of times that the control exits the node (outflow). The structural constraints of this example are:

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

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

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

$$0 \leq x_3 \leq 10x_1.$$  \hfill (10)

The functionality constraints can also be used to specify other path information. For example, we observe that the else statement ($B_2$) can be executed at most once inside the loop. This information can be specified as:

$$x_5 \leq 1x_1.$$  \hfill (11)

All the above constraints bound all the feasible values of $x_i$’s. We can solve the WCET of the code fragment by maximizing Eq. (1) subject to these constraints.

3.2. Microarchitecture Modeling

Microarchitecture modeling models the CPU pipeline and cache memory. The pipeline modeling is done during the analysis of basic block costs ($c_i$’s) [6]. In this subsection, we will describe how the direct mapped instruction cache is modeled. This is done by modifying the cost function (1) and by adding a list of linear constraints, denoted as cache constraints, which bound the cache behavior.

3.2.1 Modified Cost Function

With cache memory, the execution time of an instruction will be different depending on whether it results in a cache hit or cache miss. 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. We need a different grouping of instructions than the basic block for this analysis. We define a new grouping, the line-block or simply l-block. A l-block is a contiguous sequence of instructions within the same basic block that are mapped to the same cache set in the instruction cache. All instructions within a l-block will always have the same cache hit/miss counts, and the same total execution counts. Suppose a basic block $B_i$ is partitioned into $n_i$ l-blocks, denoted as $B_{i1}, B_{i2}, \ldots, B_{in_i}$.

For any two l-blocks mapped to the same cache set, they conflict with each other if their address tags [3] are different. Otherwise, they are called non-conflicting l-blocks.

Since l-block $B_{ij}$ 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 $B_{ij}$ are denoted as $x_{i,j}^{hit}$ and $x_{i,j}^{miss}$ respectively, and

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

The new execution time (cost function) of the program is:

$$\text{Total execution time} = \sum_i \sum_j c_i^{hit} x_{i,j}^{hit} + c_i^{miss} x_{i,j}^{miss}.$$  \hfill (13)

Here, $c_i^{hit}$ and $c_i^{miss}$ are the hit cost and the miss cost of the l-block $B_{i,j}$ respectively. $c_i^{hit}$ is the execution time spent in the CPU. For pipelined CPUs, it is equal to the sum of effective execution time of the instructions in l-block $B_{i,j}$. $c_i^{miss}$ is equal to $c_i^{hit}$ plus the penalty due to cache miss.

Equation (12) links the new cost function (13) with the program structural constraints and the program functionality constraints, both of which remain unchanged. In addition, the cache activities can now be specified in terms of the new variables $x_{i,j}^{hit}$’s and $x_{i,j}^{miss}$’s.

3.2.2 Direct Mapped Instruction Cache Analysis

Consider a simple case. For each cache set, if there is only one l-block $B_{k,l}$ mapped to it, then only its first execution may result in a cache miss, therefore,

$$x_{k,l}^{miss} \leq 1.$$  \hfill (14)

When a cache set contains two or more conflicting l-blocks, the hit/miss counts of all the l-blocks mapped to this set will be affected by the sequence in which these l-blocks are executed. An important observation is that the execution of any other l-blocks mapped to other cache sets will have no effect on these counts. This leads us to examine the control flow of the l-blocks mapped to that particular cache set by using the 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 same 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 $l$-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 set 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 $B_{k,l}$ 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 $B_{i,j}$:

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

(15)

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:

$$\sum_{u,v} p_{(s,u,v)} = 1.$$  
(16)

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

$$p_{(i,j,i,j)} \leq x_{i,j}^\text{hit} \leq p_{(i,j,i,j)} + p_{(i,j,i,j)}.$$  
(17)

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

$$x_{i,j}^\text{hit} = p_{(i,j,i,j)}.$$  
(18)

Equations (14) through (18) 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. The CCGs are network flow graphs and thus the cache constraints are typically solved rapidly by the ILP solver. For programs with function calls, the functions are treated as if they are inlined [6].

4. Set Associative Instruction Cache Analysis

The analysis of set associative instruction caches is built on the technique developed for the direct mapped case. Our objective is to determine the number of hits and misses of each $l$-block. Therefore, the cost function (13) is unchanged. As in the direct mapped instruction cache analysis, the structural and functionality constraints are unchanged. This follows because these constraints are used to model the feasible control flow, which is independent of the underlying cache configuration.

The cache constraints described in the previous section need to be replaced because they are tailored specifically for direct mapped instruction cache only. For an $n$-way set associative instruction cache, we need to keep track of the contents in the cache sets before we can determine whether the execution of the subsequent $l$-block will result in a cache hit or not. We also need to determine that if it is a miss, which line is chosen to replace. Clearly the CCGs, which only store the previous executed $l$-blocks in their nodes, are inadequate in providing the required information. Without the loss of generality, we will describe how to model 2-way set associative instruction cache with least recently used replacement policy in the following subsection. The extension to higher degrees of associativity and also other replacement policies is obvious.

4.1 States of a Cache Set

So far we have been looking at the problem from the control flow point of view, i.e., what the possible control flows among the $l$-blocks are, and how they result in cache hits/misses. Let’s look at the problem from the cache memory point of view: What are the possible cache states during the execution of the program? And how do the transitions of these cache states relate to the cache hits/misses of $l$-blocks?
Figure 4. The CSTG represents all the possible cache state transitions of a cache set during the execution of the program. It is constructed by tracing the CCG, which in this example, is constructed when l-blocks B_{i,1}, B_{4,1} and B_{5,1} in Fig. 1 conflict with each other.

Suppose we have 3 conflicting l-blocks B_{i,j}, B_{k,l} and B_{m,n} mapped to the same cache set. For 2-way set associative instruction cache, each set contains two cache lines, so the possible states of this cache set are:

[X,X], [X,B_{i,j}], [X,B_{k,l}], [X,B_{m,n}], [B_{i,j}, B_{k,l}], [B_{i,j}, B_{m,n}], [B_{i,j}, B_{k,l}], [B_{k,l}, B_{m,n}], [B_{m,n}, B_{i,j}], and [B_{m,n}, B_{k,l}].

Each state has two entries representing the contents of the two cache lines. The left most entry represents the least recently used l-block and the right most entry represents the most recently used l-block. An ‘X’ means that the entry is unknown and it may be empty. In general, if there are n cache lines in the cache set, and there are m attacking l-blocks mapping to this set, there will be potentially \( \sum_{i=0}^{m} \binom{m}{i} \) cache states.

Having defined the above cache states, let us examine the transitions between them. These transitions can best be shown by using a graph, which is called cache state transition graph (CSTG). The nodes of the CSTG are the possible cache states during the execution of the program. A possible transition from one state to another is represented by a directed edge.

A CSTG is drawn by using the CCG. Consider an example, in Fig. 1, suppose that B_1, B_3 and B_5 are partitioned into a set of l-blocks and that l-blocks B_{1,1}, B_{4,1} and B_{5,1} are mapped to the same cache set and they conflict with each other. The CCG for this cache set is shown in Fig. 4(i). For 2-way set associative instruction cache, the corresponding CSTG is shown in Fig. 4(ii). The transition in CSTG starts at state [X,X]. Since there is a flow from node ‘s’ to node ‘B_{1,1}’ in CCG, there is a state transition from [X,X] to [X,B_{1,1}] and an edge is drawn from [X,X] to [X,B_{1,1}] in the CSTG. After this state is reached, we can look at the possible outflow from node B_{1,1} in CCG and construct the corresponding transitions from [X,B_{1,1}] in CSTG. In this case, we may have a transition from [X,B_{1,1}] to either [B_{1,1}, B_{4,1}] or [B_{1,1}, B_{5,1}]. They represent the execution of either l-block B_{4,1} or l-block B_{5,1} respectively. When the program stops, a special transition is made to reach the end node ‘e’ in CSTG. In general, not all of the states may actually be present in the CSTG. In the example, [X,B_{1,1}] is not present in the CSTG.

The CSTG can be constructed for an n-way associative instruction cache. Note that if n is equal to 1, i.e., for direct mapped instruction cache, CSTG collapses to CCG.

### 4.2. New Cache Constraints

The construction of cache constraints for set associative instruction cache analysis is very similar to that for direct mapped instruction cache analysis. For 2-way set associative instruction cache, if there are \( m \) conflicting l-blocks mapping to the same set of cache lines and \( n \leq 2 \), then once these l-blocks are loaded into the cache, they will not be replaced. Therefore, there will be at most \( m \) cache misses.

If \( m > 2 \), a CCG and a CSTG for that set of cache lines are constructed. It follows directly that since CSTG is a network flow graph, a set of linear constraints can again be formulated from this graph. For every transition, say from state \([B_{i,j}, B_{k,l}]\) to state \([B_{m,n}, B_{l,m,n}]\), we assign a variable \( P([i,j,k,l], [k,l,m,n]) \) to the edge to represent the number of times this transition is made. Note that this transition occurs due to the execution of l-block \( B_{m,n} \). Therefore, the total execution count of \( B_{m,n} \) is equal to the total sum of inflows going into states with \( B_{m,n} \) in the right most line entry:

\[
\sum_{x \in \mathbb{Z}} P([x,y,z], [y,z,m,n]) = 0
\]  

The above equation links the CSTG with the CFG.

At each node \([B_{i,j}, B_{k,l}]\) in CSTG, sum of inflow must be equal to sum of outflow:

\[
\sum_{u,v} P([u,v,i,j], [i,j,k,l]) = \sum_{y,z} P([i,j,k,l], [k,l,y,z])
\]

The starting condition is:

\[
\sum_{u,v} P([X,X], [X,u,v]) = 1
\]

Finally, the lower bound of the cache hit counts is given by the following inequality:

\[
x_{hit}^{\min} \geq \sum_{u,v} P([u,v,x,y,z], [u,v,y,z]) + \sum_{u,v} P([y,z,u,v], [u,x,y,z])
\]

Here, the first term represents a self loop in the CSTG. The cache state is unchanged after the execution of l-block \( B_{x,y,z} \). The second term represents that before the execution, the cache state contains the contents of \( B_{x,y,z} \) in the cache line other than the most recently used line. The execution of \( B_{x,y,z} \) will result in a cache hit and it will also change to cache state so that the contents of \( B_{x,y,z} \) are in the most recently used line after the execution.
The above equation represents the lower bound of the cache hit counts. Like the direct mapped analysis, we need to consider the situation that the starting cache entries \((X,X)\) may contain the cache contents of the l-block \(B_{i,j}\) and therefore, \(p(X,X), [X,i,j])\) may also be counted as cache hits. For transition from \([X,X]\) to \([X,B_{i,j}]\), if there exists an exit state (state that has edge to the end node in the CSTG) \([B_{i,j}, B_{x,z}]\) or \([B_{u,v}, B_{i,j}]\) for any \(B_{u,v}\), then the transition may be a cache hit. Similarly, for transition from \([X,B_{i,j}]\) to \([B_{i,j}, B_{x,k}]\), if there exist an exit state \([B_{i,j}, B_{u,l}]\) or \([B_{k,l}, B_{i,j}]\), then the transition may also be a cache hit.

We can use CSTG to analyze direct mapped instruction cache, the cache constraints obtained from CSTG are exactly the same as those obtained in the previous section.

5. Data Cache Analysis

The objective of the data cache analysis is to determine the number of data cache hits and misses. We need to identify all instructions that will fetch their operands from main memory. Without loss of generality, we will discuss the data cache analysis method in terms of load/store instructions in RISC architectures only. This method directly applies to CISC architecture too.

The data cache analysis is different from the instruction cache analysis in two main areas. The first is that the absolute data addresses of load/store instructions are much more difficult to be determined statically than the instruction addresses. Data flow analysis is required and we have to restrict the program to have no dynamic data structures. In some cases, the data address may be impossible to determine or it may be determined to be within a certain range of values only. When we encounter such situation, simply invalidating all the data cache contents is too pessimistic. We will describe a better way to handle this situation using linear constraints.

The second difference is that unlike the instruction address, which remains unchanged, the data address of the load/store instruction may change during the execution of the program. A simple example is the elements of a data array that are accessed sequentially in a loop. At each iteration, the data address is incremented. As a result, the elements may be mapped to a number of data cache sets. And each cache set may have different cache hit/miss activities.

5.1. Two Levels of Analysis

The data cache analysis can be divided into two levels of analysis. This first is data flow analysis. This analyzes the data flow of the program and determines the absolute data addresses of load/store instructions. The data address here may denote a range of possible values, or in the case of loop statements, a list of addresses that the load/store instructions will access sequentially. This can be done by computing the use-definition chain [1] of the program. The stack frame problem can also be solved if each function call has its own instance. Overall, this is not an easy task. The accuracy of the analysis will have a direct impact on the tightness of the worst case timing analysis.

The second level of analysis is data cache conflict analysis. This collects the data addresses from data flow analysis and constructs the cache conflict graphs and linear constraints that bound the possible data hit and miss counts.

Although the data flow analysis is difficult to implement, its underlying algorithms have been well established. Our main contribution is in data cache conflict analysis, which will be described in the following subsection.

5.2. Modified Cost Function

Each execution of load/store instruction will result in either a data cache hit or a data cache miss. For each load/store instruction with instruction address \(addr\) and located in basic block \(B_i\), we assign variables \(m^{hit}_{addr}\) and \(m^{miss}_{addr}\) to denote the execution counts of this instruction that results in data cache hits and misses respectively. We have the following equation:

\[ x_i = m^{hit}_{addr} + m^{miss}_{addr}. \]  

(23)

The execution time of the program can be represented by the equation:

\[
\text{Exec. time} = \sum_i \sum_j (c^{hit}_{i,j} x_{i,j} + c^{miss}_{i,j} x_{i,j}) + \sum_{addr} (d^{hit}_{addr} m^{hit}_{addr} + d^{miss}_{addr} m^{miss}_{addr}).
\]  

(24)

Here, the first term represents the execution time spent in the execution unit and the instruction cache. It is identical to the Eq. (13) used in the instruction cache analysis, except that the constants \(c^{hit}_{i,j}\) and \(c^{miss}_{i,j}\) are now replaced by constants \(d^{hit}_{addr}\) and \(d^{miss}_{addr}\) to explicitly state that they are involved with the instruction cache. Their values are unchanged. The second term of the above equation represents the time spent in the data cache. \(d^{hit}_{addr}\) and \(d^{miss}_{addr}\) are constants representing the data cache penalties for a data cache hit and miss respectively. Unlike \(c^{hit}_{i,j}\) and \(c^{miss}_{i,j}\), \(d^{hit}_{addr}\) and \(d^{miss}_{addr}\) represent data cache penalties only. The time to execute the load/store instruction in the processor’s execution unit, including time to compute the absolute data address, bus transfer, etc., is already included in the corresponding \(c^{hit}_{i,j}\) and \(c^{miss}_{i,j}\) values.

5.3. Data Cache Conflict Graph

From data flow analysis, we obtain for each load/store instruction, a set of its possible data addresses. We can then determine the cache set each of these addresses will be mapped to. The next step is to consider the control flow
Figure 6. Data cache conflict graph for data cache set 0. Array element array[0] maps to this cache set. The other 9 data CCGs are all identical to this one except the variable names.

among the load/store instructions where their load/store addresses are mapped to the same cache set. This is very similar to the instruction cache analysis and therefore, cache conflict graph is used. Let us consider the small example shown in Fig. 5, which shows a code fragment in both source level and pseudo-assembly code. The code fragment increments the array element sequentially.

The loop body (basic block B2) contains a load instruction and a store instruction. By doing data flow analysis, we determine that these two instructions will access memory locations from 0x100 to 0x124 sequentially. If we assume that address 0x100 will be mapped to data cache set 0 and the size of each cache line is 4 bytes, then the above memory access pattern will span from data cache set 0 to data cache set 9.

The activity of each data cache set is the same. It involves loading the value of array[i] into data cache, and then later storing it back to main memory. A data cache conflict graph is drawn for each cache set. It is shown in Fig. 6. The graph is similar to the cache conflict graph in instruction cache analysis, each node has a load/store instruction. The data cache conflict graph captures the control flow of those load/store instructions that may access data addresses mapped to this data cache set.

For direct mapped data cache analysis, a $p_k$ variable is attached to each edge of the data CCG to denote the number of times the control will follow that edge. Each node contains a variable of the form $m_{addr_i, addr_j}$ where $addr_i$ denotes the instruction address of the instruction and $addr_j$ denotes the data address. Since each load/store instruction may have different data address during the execution of the program, it may have several variables in the form $m_{addr_1, addr_j}$, $m_{addr_2, addr_j}$, etc. The sum of all these variables is equal to the execution count basic block where the instruction resides in.

Having assigned the variables, we then construct linear constraints from the data CCG. The first set of constraints are from the network flow property of the data CCG. At each node of the data CCG, the sum of control flow going into a node must be equal to the sum of control flow coming out from that node, and this must also be equal to the execution count of the instruction with that particular data address inside the node. For example, the linear constraints derived from the data CCG shown in Fig. 6 are:

\begin{align}
  m_{0\times014.0\times100} &= p_1 = p_2 \quad (25) \\
  m_{0\times01c.0\times100} &= p_2 = p_3 \quad (26) \\
  p_1 &= 1 \quad (27)
\end{align}

The second set of linear constraints bounds the execution count of each load/store instruction instance. The bound is determined from the data flow analysis. In the example shown above, each load instruction instance will be executed once every time the control enters the loop. Therefore, the bound is

\begin{align}
  m_{0\times014.0\times100} &= x_1 \quad (28) \\
  m_{0\times01c.0\times100} &= x_1 \quad (29)
\end{align}

A key point here is that in some cases, we may not be able to determine the exact load/store address. Suppose that we can only determine that the load address for the first iteration is within a certain range of values, say it may be 0x100 or 0x104, then there will be a node on the data CCG 0 and the data CCG 1. And equation (28) will become:

\begin{align}
  m_{0\times014.0\times100} + m_{0\times014.0\times104} &= x_1 \quad (30)
\end{align}

This above constraint describes that either one of the data addresses will be accessed, but not both. The ILP solver will determine the worst case scenario and the corresponding value for each variable. This approach is less pessimistic than Hut's approach, which will invalidates both data cache sets in this case.

The sum of execution count of all load/store instruction instance must be equal to the execution count of the corresponding load/store instruction, which is also equal to the execution count of the corresponding basic block.

\begin{align}
  \sum_{addr_j} m_{0\times014.0\times100} &= x_2 \quad (31) \\
  \sum_{addr_j} m_{0\times01c.0\times100} &= x_2 \quad (32)
\end{align}

The hit and miss counts for each $m_{addr_i, addr_j}$ variable can be formulated from the data CCGs. For load instruction, we look at the corresponding node in the data CCG. All self loops will result in data cache hits. In addition to that, all edges going into that node from nodes where their data addresses are the same will also result in cache hits. Once we formulate the linear constraints for each $m_{hit, addr_i, addr_j}$, $m_{hit, addr_i}$ is given by:

\begin{align}
  m_{hit, addr_i} &= \sum_{addr_j} m_{hit, addr_i, addr_j} \quad (33)
\end{align}

For store instructions, the hit/miss count depends on the write policy (write through or write back) and on the policy on a write back miss (write allocate or no write allocate).
Generally, write-through caches use no write allocate and write-back caches use write allocate.

If the data cache is a write-through cache with no write allocate, then all the store instructions will write the data to main memory and cache memory. This normally takes a constant time regardless of whether the cache contains the data. Hence, for this type of cache, we do not need to consider the store instructions. The CCGs will only have nodes for load instructions.

If the data cache is a write-back cache with write allocate, then all edges going into the store node from other conflicting nodes, will represent cache misses, as the write allocate policy will load the data to cache memory. All self loops are hits. And all edges going out from the node to other conflicting nodes will represent cache misses again, because of the write-back policy.

In our example, if we assume a write-back cache with write allocate, then the linear constraints for the hit/miss counts from data CCG 0 are:

$$m_{0x014.0x100}^{hit} \leq p_1$$  \hspace{1cm} (34)

$$m_{0x01c.0x100}^{miss} = p_3$$  \hspace{1cm} (35)

### 6. Set Associative Data Cache and Unified Cache

Once we have the data CCGs, we can also analyze the set associative data cache by using techniques we developed for the set associative instruction cache. Further, we can also analyze unified cache by placing the conflicting l-blocks and load/store instructions in the same CCG. The cache constraints are then constructed based on the same principles.

### 7. Implementation

We have so far implemented the set associative instruction cache analysis in a tool cinderella$^2$. Cinderella first reads the program's binary code and builds CFGs, CCGs and CSTGs. It then outputs annotated files where the basic block variables are labeled along with the source code and asks the user to provide loop bounds. After that, cinderella can start computing the estimated WCET. The user can provide additional functionality constraints so as to eliminate some infeasible program paths and obtain tighter estimated WCET.

Unlike other tools, cinderella is source-level independent and it does not require special in-house compiler to generate additional information needed for timing analysis. In addition, we have re-written the tool so that it now has a target-dependent back end containing the binary file module, the instruction set decoding module and instruction timing module. We have written modules for modeling Intel i960KB processor and the Motorola MC68000 processor and are currently implementing data cache and unified cache analysis. By writing the above modules, one can easily port cinderella to analyze the execution times of programs running on other hardware platforms.

### 8. Experimental Results

We have demonstrated that cinderella can determine tight estimated WCET for programs running on an Intel i960KB processor, which has an on-chip 512-byte direct mapped instruction cache [6]. In this paper, we would like to focus more specifically in the accuracy of cache analysis. For this reason, we assume that each cache hit will take 0 clock cycle and each cache miss will take 1 clock cycle. All other execution times spent in the CPU are assumed to be zero. Given these assumptions, the estimated WCET computed by cinderella is equal to the worst case number of cache misses. Table 1 shows the benchmark programs.

Since it is impractical to simulate all the possible combinations of input data values and starting states of the system, a program’s actual WCET cannot be determined easily. We study each program carefully and try to identify the worst case input data values. For programs des and dppeg, we run each program with a number of different random input data and choose the one with the longest execution time as the worst case input data. Each program is executed with this worst case input data and its instruction

---

$^2$Details of the tool and the benchmark programs can be obtained from [http://www.princeton.edu/~vaull/cinderella](http://www.princeton.edu/~vaull/cinderella).
Table 1. Set of benchmark programs, their descriptions, source file line sizes and the binary code sizes.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Lines</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>Circle drawing routine</td>
<td>100</td>
<td>1,588</td>
</tr>
<tr>
<td>des</td>
<td>Data Encryption Standard</td>
<td>192</td>
<td>1,852</td>
</tr>
<tr>
<td>dhrty</td>
<td>Dhrystone benchmark</td>
<td>761</td>
<td>1,360</td>
</tr>
<tr>
<td>djjpeg</td>
<td>JPEG decompression (128x96 color image)</td>
<td>857</td>
<td>5,408</td>
</tr>
<tr>
<td>fdct</td>
<td>JPEG forward discrete cosine transform</td>
<td>300</td>
<td>996</td>
</tr>
<tr>
<td>line</td>
<td>Line drawing routine</td>
<td>165</td>
<td>1,555</td>
</tr>
<tr>
<td>stats</td>
<td>Calculate the sum, mean and variance of two arrays [2]</td>
<td>100</td>
<td>656</td>
</tr>
<tr>
<td>stats2</td>
<td>Same as stats, but with inlined functions</td>
<td>90</td>
<td>596</td>
</tr>
<tr>
<td>whetstone</td>
<td>Whetstone benchmark</td>
<td>196</td>
<td>2,760</td>
</tr>
</tbody>
</table>

memory trace is fed into the dineroIII\textsuperscript{3} cache simulator so as to obtain the worst case number of cache misses. We assume that this dineroIII result is very close to the actual worst case number of cache misses and it is used to validate cinderella’s estimated WCET.

Table 2 compares cinderella’s estimates with the dineroIII simulation results under different instruction cache settings. It also shows the performance of solving the ILP problems. For each table, the second column shows the estimated WCET computed by cinderella and the third column shows the simulated cache misses done by dineroIII. All estimated WCETs bound their corresponding simulation results tightly. For most programs, their estimated WCETs are identical to the simulation results. This shows that with sufficient functionality constraints, cinderella can compute extremely tight estimated WCET. For programs des and djjpeg, the differences are due to the existence of complex data dependent program paths, which cannot be easily determined and modeled by the functionality constraints. This is especially true for program djjpeg, where the symbols in the JPEG image can typically be compressed during the Huffman encoding process [11]. But in our worst case analysis, we have to assume that these symbols are so randomly distributed that no compression is achieved in the worst case scenario. This results in the larger pessimism in the estimation.

The next four columns of each table show the number of variables, which are broken down in (i) $d$‘s, the CFG edge variables, (ii) $f$‘s, the function call edge variables, (iii) $p$‘s, the CSTG edge variables, and (iv) $x$‘s, the basic block and $l$-block variables. The following three columns show the number of structural, cache and functionality constraints. Each program may have more than one functionality constraint set, such that at least one set of the functionality constraints must be satisfied in determining the estimated WCET [6]. Most benchmark programs have only one set of functionality constraints, except des and dhrty, which have two and eight sets respectively. Each set of function-

\textsuperscript{3}DineroIII is written by Mark D. Hill and more information can be obtained from http://www.cs.wisc.edu/~larus/warts.html.

\textsuperscript{4}lp.solver is written by Michel Berkelaar and can be retrieved at ftp://ftp.es.ele.tue.nl/pub/lp.solver.

9. Conclusions and Future Work

We have described a method to determine a tight estimated worst case execution time of a program running on a hardware system with cache memory. The method is based on integer linear program formulation. Its advantages are:

- The program path analysis and the microarchitecture modeling can be handled independently. The structural constraints and the functionality constraints are unchanged even when we alter the underlying microarchitecture analysis to model different cache configurations.
- Because the worst case execution time is solved by determining the worst case execution counts of the blocks, no explicit path enumeration is required. This reduces the solution search space significantly, while maintaining the minimum sufficient information for
Table 2. Results of instruction cache analysis with different cache parameters.

(i) 512-byte direct mapped instruction cache with 16-byte line size.

<table>
<thead>
<tr>
<th>Program</th>
<th>Cinderella</th>
<th>Dinosaur</th>
<th>Variables</th>
<th>Constraints</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>458</td>
<td>443</td>
<td>8</td>
<td>1</td>
<td>81 18</td>
</tr>
<tr>
<td>des</td>
<td>4,188</td>
<td>3,872</td>
<td>174 11</td>
<td>723 220</td>
<td>342 1,097</td>
</tr>
<tr>
<td>dhry</td>
<td>8,304</td>
<td>8,304</td>
<td>102 21</td>
<td>490 214</td>
<td>289 779</td>
</tr>
<tr>
<td>jpeg</td>
<td>316,394</td>
<td>230,861</td>
<td>296 20</td>
<td>1,816,416</td>
<td>613 2,568</td>
</tr>
<tr>
<td>fdct</td>
<td>63</td>
<td>63</td>
<td>8</td>
<td>0</td>
<td>18 12</td>
</tr>
<tr>
<td>line</td>
<td>101</td>
<td>99</td>
<td>31</td>
<td>2</td>
<td>264 51</td>
</tr>
<tr>
<td>stats</td>
<td>47</td>
<td>47</td>
<td>28</td>
<td>13</td>
<td>57 12</td>
</tr>
<tr>
<td>stats2</td>
<td>44</td>
<td>44</td>
<td>28</td>
<td>7</td>
<td>41 54</td>
</tr>
<tr>
<td>whetstone</td>
<td>18,678</td>
<td>18,678</td>
<td>52</td>
<td>3</td>
<td>238 76</td>
</tr>
</tbody>
</table>

(ii) 512-byte 2-way set associative instruction cache with 16-byte line size.

<table>
<thead>
<tr>
<th>Program</th>
<th>Cinderella</th>
<th>Dinosaur</th>
<th>Variables</th>
<th>Constraints</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>482</td>
<td>473</td>
<td>8</td>
<td>1</td>
<td>127 18</td>
</tr>
<tr>
<td>des</td>
<td>1,116</td>
<td>1,085</td>
<td>174 11</td>
<td>1,095 220</td>
<td>342 1,217</td>
</tr>
<tr>
<td>dhry</td>
<td>8,002</td>
<td>8,002</td>
<td>102 21</td>
<td>612 214</td>
<td>289 823</td>
</tr>
<tr>
<td>jpeg</td>
<td>380,471</td>
<td>278,869</td>
<td>296 20</td>
<td>4,814,416</td>
<td>613 3,189</td>
</tr>
<tr>
<td>fdct</td>
<td>63</td>
<td>63</td>
<td>8</td>
<td>0</td>
<td>41 12</td>
</tr>
<tr>
<td>line</td>
<td>64</td>
<td>63</td>
<td>31</td>
<td>2</td>
<td>330 51</td>
</tr>
<tr>
<td>stats</td>
<td>51</td>
<td>51</td>
<td>28</td>
<td>13</td>
<td>130 72</td>
</tr>
<tr>
<td>stats2</td>
<td>44</td>
<td>44</td>
<td>28</td>
<td>7</td>
<td>112 54</td>
</tr>
<tr>
<td>whetstone</td>
<td>144</td>
<td>144</td>
<td>52</td>
<td>3</td>
<td>313 76</td>
</tr>
</tbody>
</table>

(iii) 1KB 2-way set associative instruction cache with 16-byte line size.

<table>
<thead>
<tr>
<th>Program</th>
<th>Cinderella</th>
<th>Dinosaur</th>
<th>Variables</th>
<th>Constraints</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>172</td>
<td>172</td>
<td>8</td>
<td>1</td>
<td>100 18</td>
</tr>
<tr>
<td>des</td>
<td>169</td>
<td>166</td>
<td>174 11</td>
<td>1,097 220</td>
<td>342 1,291</td>
</tr>
<tr>
<td>dhry</td>
<td>4,933</td>
<td>4,933</td>
<td>102 21</td>
<td>715 214</td>
<td>289 834</td>
</tr>
<tr>
<td>jpeg</td>
<td>43,498</td>
<td>42,451</td>
<td>296 20</td>
<td>4,652,416</td>
<td>613 3,447</td>
</tr>
<tr>
<td>fdct</td>
<td>63</td>
<td>63</td>
<td>8</td>
<td>0</td>
<td>18 12</td>
</tr>
<tr>
<td>line</td>
<td>64</td>
<td>63</td>
<td>31</td>
<td>2</td>
<td>302 51</td>
</tr>
<tr>
<td>stats</td>
<td>44</td>
<td>44</td>
<td>28</td>
<td>13</td>
<td>57 72</td>
</tr>
<tr>
<td>stats2</td>
<td>38</td>
<td>38</td>
<td>28</td>
<td>7</td>
<td>41 54</td>
</tr>
<tr>
<td>whetstone</td>
<td>144</td>
<td>144</td>
<td>52</td>
<td>3</td>
<td>266 76</td>
</tr>
</tbody>
</table>

^a Two sets of functionality constraints. Each of them has 16 constraints.

^b Eight sets of functionality constraints. Four of them have 24 constraints each and the other four have 26 constraints each.

From our experimental results, we see that our method is both fast and accurate. The estimated WCETs computed by Cinderella tightly bound the simulated WCETs. We are currently implementing the data cache analysis part and are also building modules for modeling other hardware platforms in a reentrant environment.

10. Acknowledgements

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

References


