Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software

Yau-Tsun Steven Li
Sharad Malik
Andrew Wolfe

Introduction

- Paper examines the problem of determining the bound on the worst case execution time (WCET) of a given program on a given processor.
- Two important issues in solving this problem:
  - Program path analysis
  - Microarchitecture modelling
- Method which address both issues is proposed

Two issues involved in solving this problem.

Program path analysis.

- This determines what sequence of instructions will be executed in the worst case scenario.
- Infeasible program paths removed from the solution search space
- done by a data flow analysis of the program
- analysis should provide a mechanism for program path annotations.

Microarchitecture modeling

- Models the hardware system and computes the WCET of a given sequence of instructions
- becoming difficult to model
- most modern processors have pipelined instruction execution units and cached memory systems.

Proposed Solution

- <u>Address both issues</u>
- determine a tight bound on a program's worst case execution time.
- Explicit path enumeration not necessity to obtain tight estimated WCET
- Method determine worst case execution count of instruction and from these counts computes the estimated WCET
- includes a direct-mapped instruction cache analysis
- uses an integer linear programming formulation to solve the problem.
- allows the user to provide program path annotations so that a tighter bound may be obtained.

Program path analysis problem handling

- Pessimistic approach:

  Used simple microarchitecture model that assumes the execution time of an instruction to be a constant, i.e., every instruction fetch is assumed to result in a cache miss.

- method uses the counting approach to compute the estimated WCET.

- method converts the problem of solving the estimated WCET

into a set of integer linear programming (ILP) problems

ILP Formulation

- Assumption: Each instruction takes a constant time to execute

- Instructions within a basic block are always executed together, their execution counts are always the same.
  - let xi be the execution count of a basic block Bi, and Ci be the execution time of the basic block,
  - given that there are N basic blocks in the program,

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

- possible values of xi are constrained by the program structure and the possible values of the program variables.

- If these constraints represented as linear inequalities, ➔ the problem of finding the estimated WCET of a program is reduced to an integer linear programming (ILP) problem

Linear Constraints

Divided into two parts:

- **Program structural constraints,**

  Derived automatically from the program's control flow graph (CFG)

- **program functionality constraints,**

  -provided by the user to specify loop bounds and other path information

  -or extracted from the program semantics**.**

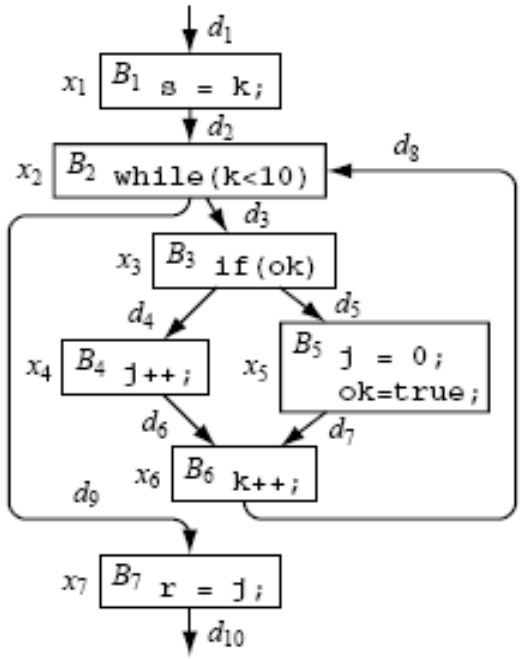
- total time required to solve the estimated WCET depends on the number of functionality constraint sets and the time to solve each constraint set.

- the complexity of solving each ILP problem, an NP-hard problem.

Example of Construction of these constraints

A conditional statement is nested inside a while loop

```
/*  k>=0 */
S=k;
While (k < 10) {if (ok)
J++;
Else {
J =0; ok=true;}
K++;
} r=j;
```



(ii) CFG

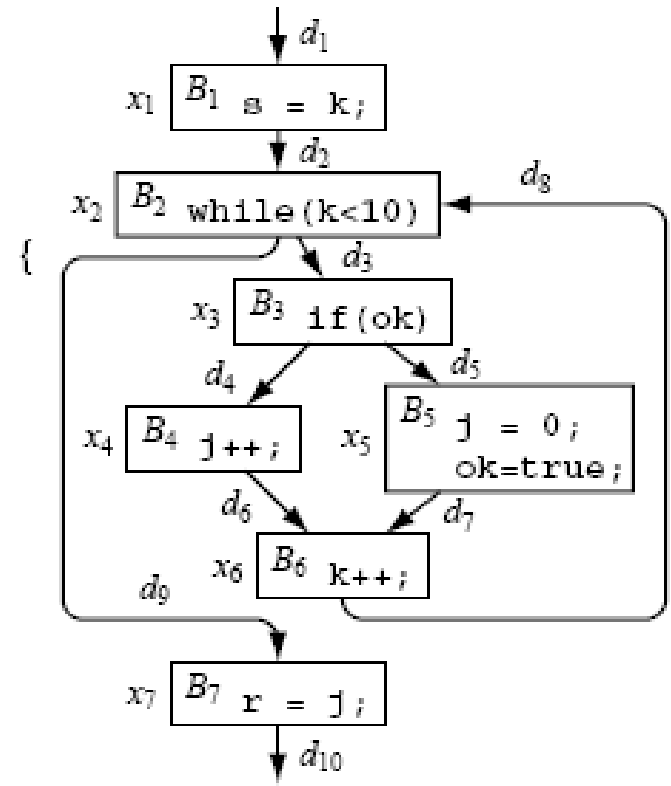
di **= a count of the the number of times that the program control passes through that edge.**

Each node in the CFG represents a basic block *Bi*.
basic block execution count, *xi*, is associated with each node.

Structural constraints

- Structural constraints can be derived from the CFG
- Fact: for each node Bi, 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).
- structural constraints of this example
- Code fragment executed once, so d1=1

$$
\begin{aligned}
d_1 &= 1 \\
x_1 &= d_1 = d_2 \\
x_2 &= d_2 + d_8 = d_3 + d_9 \\
x_3 &= d_3 = d_4 + d_5 \\
x_4 &= d_4 = d_6 \\
x_5 &= d_5 = d_7 \\
x_6 &= d_6 + d_7 = d_8 \\
x_7 &= d_9 = d_{10}.
\end{aligned}
$$



- structural constraints do not provide any loop bound information

Functional constraints

- Loop bound information can be provided by the user as a functionality constraint.

Example: since k is positive before it enters the loop, the loop body will be executed between 0 and 10 times each time the loop is entered

The constraints to specify this information are: $0x_1 \leq x_3 \leq 10x_1,$

- The functionality constraints can also be used to specify other path information.

Example: the else statement (*B*5) can be executed at most once inside the loop.

This information can be specified as:

$$x_5 \leq 1x_1.$$

- To solve the estimated WCET, each set of the functionality constraint sets is combined (the conjunction taken) with the set of structural constraints.
- The combined set is passed to the ILP solver with cost function to be maximized.

Microarchitecture Modeling

- Previously, the modeling was simple because the execution time of an instruction was largely independent of others

- goal is to model the CPU pipeline and the cache memory systems and find out the execution times (*Ci*) of the basic Blocks

- Method limited to model a direct-mapped instruction cache.

- can be extended to handle set associative instruction cache memory.

Direct-mapped Instruction Cache Analysis

- To incorporate cache memory analysis in ILP model
  - need to modify the cost function
  - add a list of linear constraints, denoted as cache constraints, representing the cache memory behavior
- Modified Cost Function
  - With cache memory execution time of an instruction will be different depending on whether it results in a cache hit or cache miss.
  - need to subdivide the original instruction counts into counts of cache hits and misses.
- If cache hit and miss count and hit and miss execution time of instruction determined then tighter bound on execution time of program is established

New type of atomic structure *line-block* (*l-block)* for analysis

- Adjacent instructions can be grouped together
- *l-block* is defined as a contiguous sequence of instructions within the same basic block that are mapped to the same line in the instruction cache.
- a basic block Bi is partitioned into ni l-blocks. We denote these l-blocks as Bi.1, Bi.2, . . . , Bi.ni .
- All instructions within an *l-block* will always have the same cache hit/miss counts, and the same total execution counts
- The cache hit and the cache miss counts of l-block Bi. j are denoted as
- Xhit i. j and xmiss i.j

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

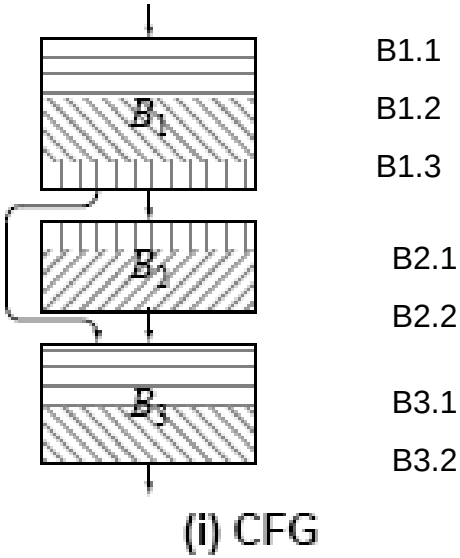
- the cache behavior can now be specified in terms of the new variables
- xhit i. j and xmiss i. j

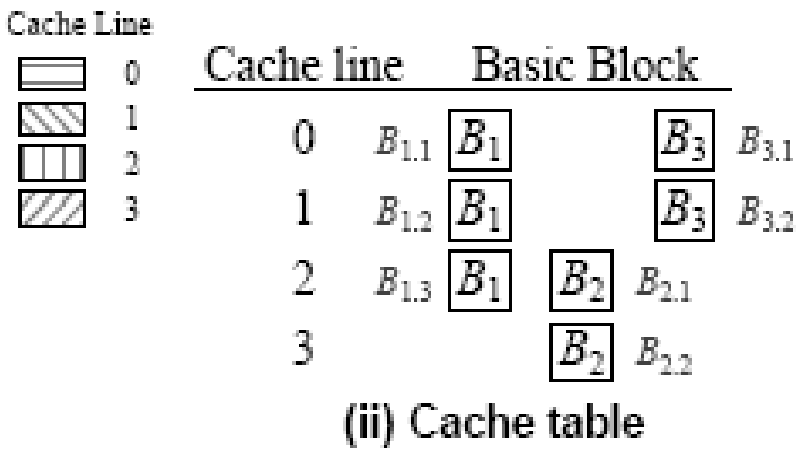
New total execution time (Cost Function)

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

Example showing how the l-blocks are constructed.

Each rectangle in the cache table represents a l-block.
CFG with 3 basic blocks and instruction has 4 cache line





•any two l-blocks that map to the same cache line, they **conflict** with each other if the execution of one l-block will displace the cache content of the other.

• Otherwise, they are called **non-conflicting** l-blocks e.g. B1.3 and B2.1

Cache Constraints

- used to constrain the hit/miss counts of the l-blocks.
- simple case : For each line only one l-block $B_{k.l}$ mapping.
  First execution of this l-block may cause a cache miss and all subsequent executions will result in cache hits.

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

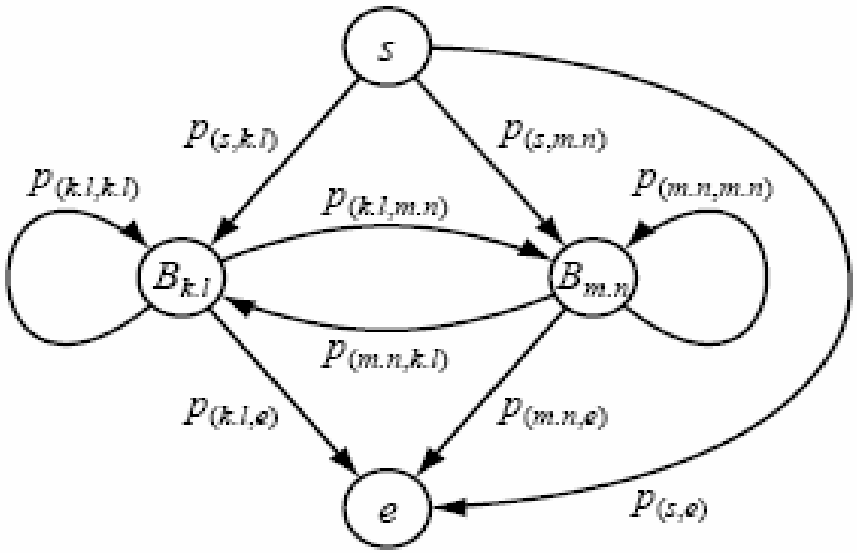
- case : Two or more **non-conflicting** l-blocks map to the same cache line($B_{1.3}$ and $B_{2.1}$
  The execution of any of them will load all the l-blocks into the cache line. sum of their cache miss counts is at most one.

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

- Case: a cache line contains two or more **conflicting** l-blocks, the hit/miss counts of all the l-blocks mapped to this line will be affected by the sequence in which these l-blocks are executed.

Cache Conflict Graph (Network flow graph)

- cache conflict graph (CCG) is constructed for every cache line containing two or more conflicting l-blocks. Example :Cache line contains 2 conflicting graph
- start node ‘*s*’, an end node ‘*e*’, and a node Bk.l for every l-block Bk.l mapped to the same cache line.
- if there exists a path in the CFG from basic block Bk to basic block Bm without passing through the basic blocks of any other l-blocks of the same cache line



Program begins at S node. i)After executing other L-block from other cache line eventually reaches to one of conflicting graph

ii) After executing Bk.l may pass other l-block and reaches to Bm.n or directly passes to Bm.n

p(i. j,u.v) to count the number of times that the control passes through that edge

continued

- At each node Bi. 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 Bi. j .
- two constraints are constructed at each node Bi. j:

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

This set of constraints is linked to structural and functionality constraints via the *x*-variables.

•Program executed once at start node $\sum_{u,v} p(s,u,v) = 1.$

variable *p*(*i*. *j*,*i*. *j*) represents the number of times that the control flows into *l-block Bi*. *j* after executing *l-block Bi*. *j*

- If both edges (Bi.j, e) and (s, Bi.j) exists then
  the program variable p (s,i. j) *may* also be counted as a cache hit.

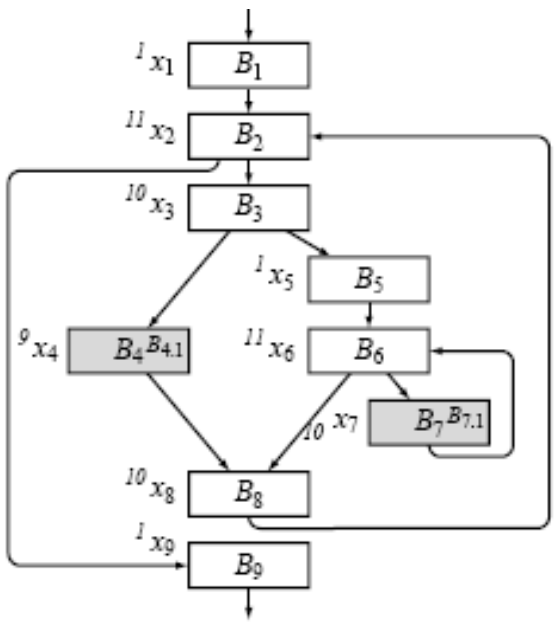
$$p_{(i.j,i.j)} \leq x_{i.j}^{hit} \leq p_{(s,i.j)} + p_{(i.j,i.j)}.$$

if any of edges (s,Bi. j) and (Bi. j ,e) does not exist, tl $x_{i.j}^{hit} = p_{(i.j,i.j)}.$

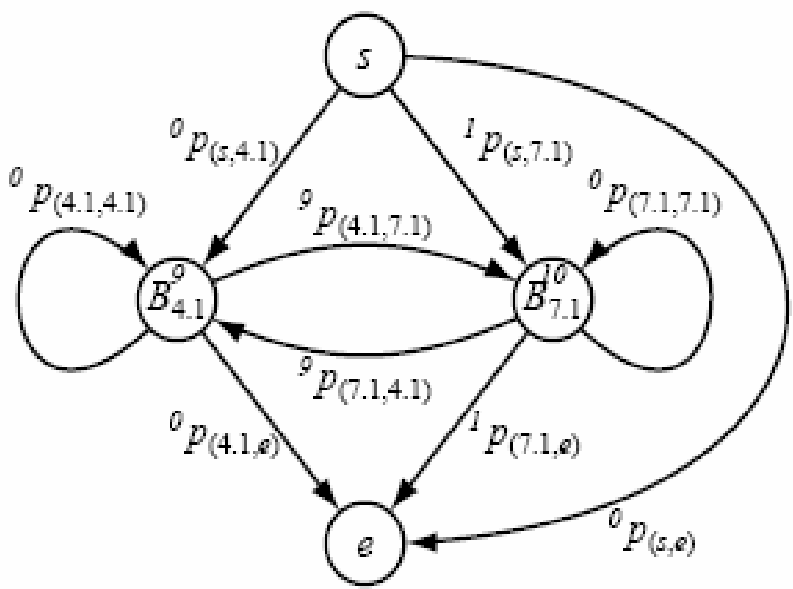
Bounds on p-variables

- some path sequencing information can be expressed in terms of p-variables as extra functionality constraints
- Without the correct bounds, the solver may return an infeasible

l-block count and an overly pessimistic estimated WCET.



(i) CFG



(ii) CCG

example showing two conflicting l-blocks (B4.1 and B7.1) from two different loops.

The italicized numbers shown on the left of the variables are the pessimistic worst case solution returned from ILP solver.

For any variable p(i. j,u.v), its bounds are: $0 \leq p_{(i.j,u.v)} \leq \min(x_i, x_u).$

A loop preheader is the basic block just before entering the loop. For instance, in the example shown in Fig. 4, basic block B1 is the loop preheader of the outer loop and basic block B5 is the loop preheader of the inner loop.

a constraint at loop preheader *B*5 is needed

$$p_{(5,7.1)} + p_{(4.1,7.1)} \leq x_5.$$

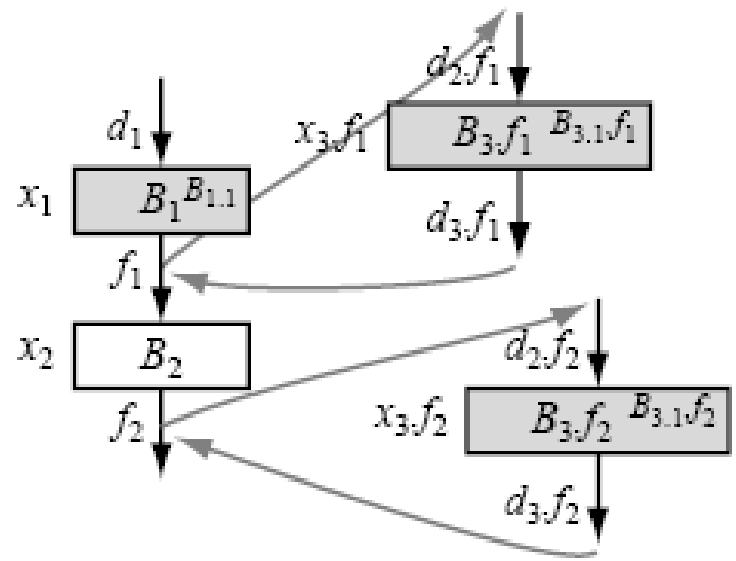
Interprocedural call

- function may be called many times from different locations of the program.
- Every function call is treated as if it is inlined.
- a function call is represented by an f - edge pointing to an instance of the callee function's CFG.
- edge has a variable 'fk ' which represents the number of times that the particular instance of the callee function is called

Function inc is called twice in the main function

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


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

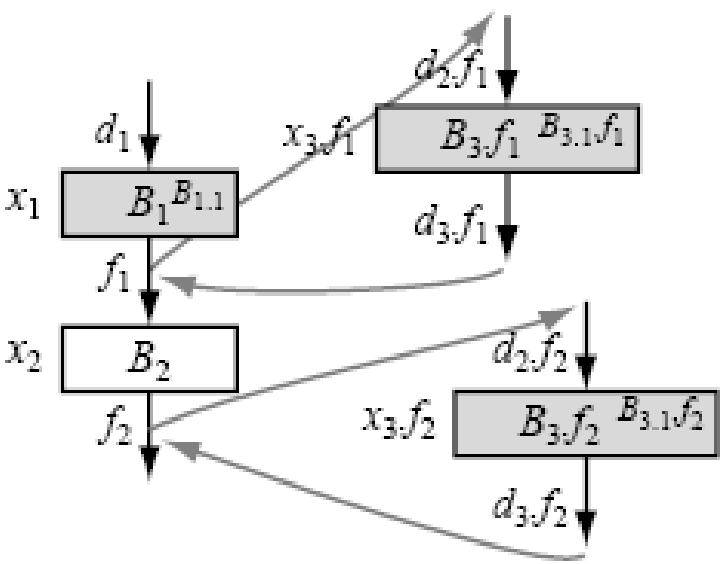


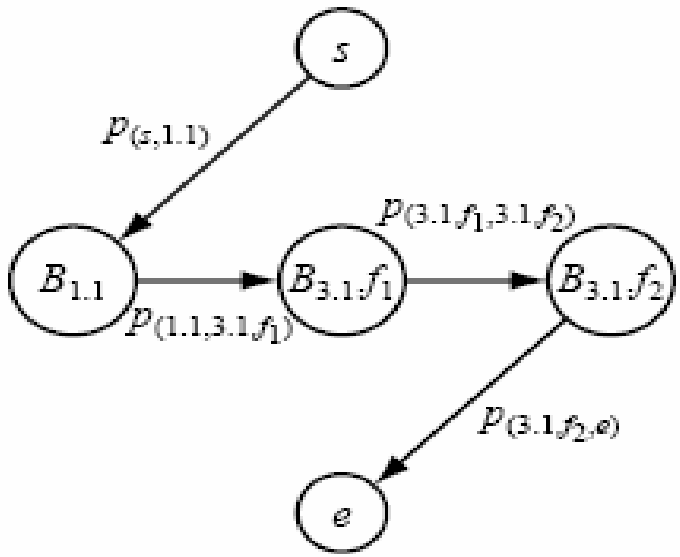
$$
\begin{aligned}
d_1 &= 1 \\
x_1 &= d_1 = f_1 \\
x_2 &= f_1 = f_2 \\
d_2.f_1 &= f_1 \\
x_3.f_1 &= d_2.f_1 = d_3.f_1 \\
d_2.f_2 &= f_2 \\
x_3.f_2 &= d_2.f_2 = d_3.f_2 \\
x_3 &= x_3.f_1 + x_3.f_2
\end{aligned}
$$

last equation above links the total execution counts of basic block *B*3 with its counts from two instances of the function

CCG

- CCG is constructed as before by treating each instance of l-block Bi. j . fk as different from other instances of the same l-block.
- In the example, if l-block B1.1 conflicts with l-block B3.1, then since l-block B3.1 has two instances ($B_{3.1}.f_{1.}$ and $B_{3.1}.f_{2.}$), there will be 5 nodes in the CCG





(iii) CCG

Cache constraints

- cache constraints and the bounds on *p* variables are constructed as before,
- the hit constraints are modified slightly. Edge going from $B_{i.j} \cdot f_{k.}$ to $B_{i.j} \cdot f_{1.}$ counted as cache hit of block *Bi.j*
- The complete cache constraints derived from the example's CCG are

$$
\begin{aligned}
x_1 &= x_{1.1}^{hit} + x_{1.1}^{miss} \\
x_2 &= x_{2.1}^{hit} + x_{2.1}^{miss} \\
x_3 &= x_{3.1}^{hit} + x_{3.1}^{miss} \\
x_{2.1}^{miss} &\leq 1 \\
x_1 &= p_{(s,1.1)} = p_{(1.1,3.1.f_1)} \\
x_{3.f_1} &= p_{(1.1,3.1.f_1)} = p_{(3.1.f_1,3.1.f_2)} \\
x_{3.f_2} &= p_{(3.1.f_1,3.1.f_2)} = p_{(3.1.f_2,e)} \\
p_{(s,1.1)} &= 1 \\
x_{1.1}^{hit} &= 0 \\
x_{3.1}^{hit} &= p_{(3.1.f_1,3.1.f_2)}.
\end{aligned}
$$

CPU Pipeline

- The CPU pipeline is considered to be relatively easy to model because it is only effected by adjacent instructions.
- As $c_{i.j}^{hit}$ and $c_{i.j}^{miss}$ must be constants

Assumption: the time required to execute a sequence of instructions in the CPU pipeline is always a constant throughout the execution of the program.

- hit cost $c_{i.j}^{hit}$ of a l-block Bi. j is determined by adding up the effective execution times of the instructions in the l-block
- 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
- Additional time is also added to the last l-block of each basic block so as to ensure that all the buffered load/store instructions are completed when the control reaches the end of the basic block.
- miss cost $c_{i.j}^{miss}$ of the l-block is equal to the time needed to load the instructions of the l-block into the cache memory and to execute them in the CPU.

Implementation

- cache analysis method has been implemented in a tool called cinderella4, which estimates the WCET of programs running
- The tool reads the subject program's executable code and constructs the CFGs and the CCG
- outputs the annotation files in which the 'x 'and 'f ' are labeled along with

the program's source code

- user is then asked to provide loop bounds
- estimated WCET can thus be computed
- user can provide additional path information, if available, to tighten this bound.

Experiment

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

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

Conclusion

- tight bound on a program's WCET is estimated.
- small amount of pessimism due to

(i) insufficient path information from the user

so that some infeasible program paths are considered,=>can be reduced by providing more path information

(i) inaccuracy in microarchitecture modeling

affects the accuracy of the values of Chiti. j and Cmissi. j =>reduced by a more sophisticated hardware model