History-based Schemes and Implicit Path Enumeration

Claire Burguière and Christine Rochange
Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier
31062 Toulouse cedex 9, France
{burguier,rochange}@irit.fr

Abstract

The Implicit Path Enumeration Technique *is often used to compute the WCET of control-intensive programs. This method does not consider execution paths as ordered sequences of basic blocks but instead as sets of basic blocks with their respective execution counts. This way of describing an execution path is adequate to compute its execution time, provided that safe individual WCETs for the blocks are known. Implicit path enumeration has also been used to analyze hardware schemes like instructions caches or branch predictors the behavior of which depends on the execution history. However, implicit paths do not completely capture the execution history since they do not express the order in which the basic blocks are executed. Then the estimated longest path might not be feasible and the estimated WCET might be overly pessimistic. This problem has been raised for cache analysis. In this paper, we show that it arises more acutely for branch prediction and we propose a solution to tighten the estimation of the misprediction counts.*

1 Introduction

The difficulty of evaluating the Worst-Case Execution Time of a real-time application comes from the – generally – huge number of possible paths that makes it intractable to analyze each of them individually. The single-path programming paradigm [9] would noticeably simplify the WCET computation but it has a cost in terms of performance that might not be acceptable. This is the reason why much research effort has been put on developing WCET evaluation approaches based on static analysis [10]. These approaches factorize the efforts by building up the WCET of the complete program from the individual WCETs of basic blocks. The Implicit Path Enumeration Technique [13], also known as *IPET*, is a very popular method for WCET calculation. It expresses the search of the WCET as an Integer Linear Programming problem where the program execution time is to be maximized under some constraints on the execution counts of the basic blocks. With this technique, an execution path is defined by the set of the executed blocks with their respective execution counts but the order in which they are executed is not expressed.

More and more complex processors are used in real-time embedded systems and it is a real challenge to take into account all of their advanced features in WCET analysis. In particular, some mechanisms have a behaviour that depends on the execution history which is difficult to capture by static analysis. These mechanisms include cache memories and dynamic branch predictors. In this paper, we consider bimodal branch prediction as an example of such schemes.

Various methods to take branch prediction into account have been proposed in the litterature. As explained below, we focus on the approach by Li *et al.* [12] that we have later extended [5] to take into account 2-bit prediction counters. In this paper, we show that both models can lead to over-estimated WCET because the worst-case number of mispredictions computed by IPET would correspond to an infeasible execution path. We show how they should be revised to only reflect feasible behaviours of the branch prediction scheme. Experimental results show that the revised model tighten the estimated WCET.

The paper is organized as follows. In Section 2, we illustrate the differences between implicit and explicit execution paths by an example. Section 3 gives an overview of dynamic branch prediction and lists previous work on branch prediction modeling for WCET analysis. We show how misprediction counts can be over-estimated in Section 4 and we propose an extended model to tighten the estimated WCET in Section 5. Section 6 concludes the paper.

2 Implicit vs. explicit execution paths

Figure 1 gives an example code that will be used throughout this paper and Figure 2 shows the corresponding CFG.

```
#define M 4
#define N 5
int main() {
  int i, j;
  int mat[M][N];
  for (i=0; i<M; i++){
     mat[i][0]=1;
     for (j=0; j<N; j++)
        mat[i][j]=i+j;
  }
}
```

Figure 1. Example code.

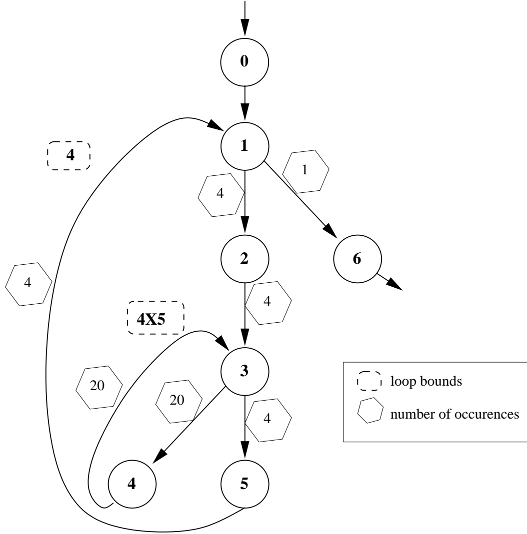

Figure 2. Example CFG.

Explicit paths. An *explicit* execution path is an ordered list of the executed basic blocks. In our example, the only possible execution path is defined by the sequence of blocks:

$$\Big(\mathtt{b1} - \mathtt{b2} - (\mathtt{b3} - \mathtt{b4})_{\times 5} - \mathtt{b3} - \mathtt{b5}\Big)_{\times 4} - \mathtt{b1} - \mathtt{b6}$$

Path-based WCET analysis [11][16] explores explicit paths but this might be costly because, as said before, a program may have many possible (explicit) paths. Some static WCET analysis methods simplify the path exploration while still considering explicit paths. For example, the Extended Timing Schema [17][15] works on the Syntax Tree.

Implicit paths. The IPET method [13] considers *implicit* paths. An implicit path is defined by the list of its basic blocks and of their execution counts. The implicit path corresponding to the explicit path given above is:

$$\big(\mathtt{b1}_{\times 5}, \mathtt{b2}_{\times 4}, \mathtt{b3}_{\times 24}, \mathtt{b4}_{\times 20}, \mathtt{b5}_{\times 4}, \mathtt{b6}_{\times 1}\big)$$

An implicit path defines many possible explicit paths but, in general, most of them are infeasible. For example, the implicit path given above could be expanded as below, where the inner loop is executed three times with a single iteration and once with 17 iterations, which is not consistent with the program semantics.

$$(\mathtt{b1} - \mathtt{b2} - \mathtt{b3} - \mathtt{b4} - \mathtt{b3} - \mathtt{b5})_{\times 3} - \mathtt{b1} - \mathtt{b2} - \\ -(\mathtt{b3} - \mathtt{b4})_{\times 17} - \mathtt{b3} - \mathtt{b5} - \mathtt{b1} - \mathtt{b6}$$

In the IPET method, the execution time of an implicit path is computed by adding the individual execution times of the basic blocks weighted by their execution counts. The WCET is obtained by maximizing the total execution time under some contraints that link the execution counts of the nodes and edges of the CFG: structural constraints directly express the CFG structure and flow constraints express loop bounds and infeasible paths.

As long as implicit paths are only used to compute a global execution time by summing individual times, there is no need to provide further information about the program semantics. However, when some mechanisms based on the execution history have to be modeled within the same framework, the *implicit* expression of execution paths might not be sufficient. This problem has been raised in the case of instruction cache analysis [14]. In this paper, our purpose is to show that it can also arise when modeling branch prediction and that it is a bit more complex in this case. However, we will provide a solution to get round it.

3 Branch prediction and WCET estimation

3.1 Bimodal branch prediction

Branch prediction enhances the pipeline performance by allowing the speculative fetching of instructions along the predicted path after a conditional branch has been encountered and until it is resolved. If the branch was mispredicted, the pipeline is flushed and the other path is fetched and executed. In the hardware *bimodal* branch predictor [18], the branch direction is predicted from a 2-bit saturating counter stored in the *Branch History Table* (which is indexed by the branch PC). If the branch is predicted as taken (counter equal to **11** or **10**), the target address is read in the *Branch Target Buffer*, otherwise the instruction fetch proceeds sequentially. When the branch is later computed, the prediction counter is updated as shown in Figure 3.

3.2 Modeling branch prediction for WCET estimation

3.2.1 Background

Modeling bimodal dynamic branch predictors for WCET analysis has been the purpose of several papers these last

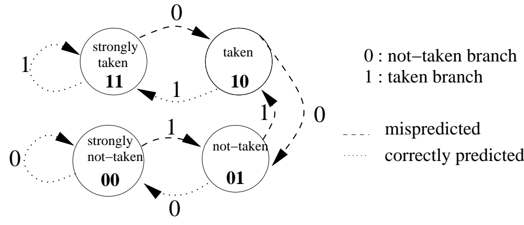

Figure 3. Bimodal branch prediction.

years. Some of the proposed techniques are *decoupled* from the pipeline analysis. Colin and Puaut [8] use static simulation to determine whether the prediction counter associated to a branch might be corrupted by another branch. Then they combine these results with an analysis of the behaviour of the 2-bit counters related to algorithmic structures to calculate bounds on the misprediction counts. Other works assume that branch aliasing can be prevented and refine the analysis of branching patterns related to algorithmic structures [1][6]. Once misprediction counts have been determined, the estimated execution time is augmented by the corresponding misprediction penalties.

To take into account tigher *per-branch* misprediction penalties, branch prediction modeling can also be *integrated* to the WCET computation with IPET [2][4]. Li *et al.* go further by completely modeling the behaviour of the branch prediction scheme within the IPET model [12]. They take conflicts in the Branch History Table into account but they only consider 1-bit prediction counters. In [5] we argue for techniques to prevent aliasing and we extend their model by considering 2-bit counters. Our discussion here is based on this last work.

Modeling branch prediction as part of WCET computation has several advantages: (1) any kind of loop can be analyzed, even if it is not well structured (*e.g.* several exit points); (2) the analysis of branches implementing conditional structures does not require any particular effort; (3) per-branch misprediction penalties can be specified.

3.2.2 Baseline model

The estimation of misprediction counts is combined to WCET computation by IPET by the way of additional constraints that: (a) express the way the prediction counters evolve; and (b) link the evolution of the prediction counters to the execution counts of the blocks and edges in the CFG. In this section, we give a simplified overview of the model. The variables used to evaluate the WCET by IPET are:

x_i	execution count of block i
$x_{i\xrightarrow{d}}$	execution count of the edge leaving block i when the branch direction is d $x_i = x_{i\xrightarrow{0}} + x_{i\xrightarrow{1}}$

The constraints added to model a bimodal branch predictor (without aliasing in the Branch History Table) use some additional variables (execution counts) presented in Figure 4.

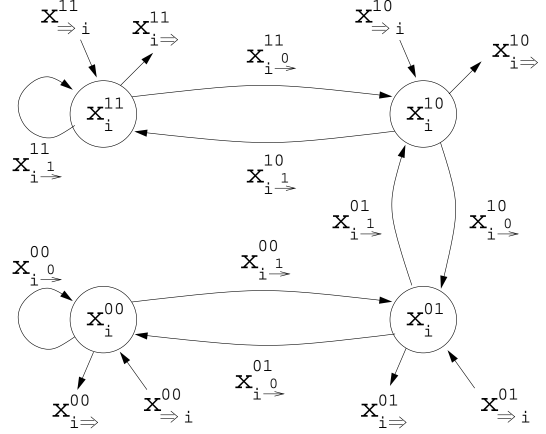

Figure 4. Variables used to model branch prediction for block i

The set of possible states for a 2-bit branch prediction counter is denoted as $\mathcal{C} = \{00, 01, 10, 11\}$ and the set of possible directions d after a branch is denoted as $\mathcal{D} = \{0, 1\}$. The constraints that model the way the prediction for the branch of block i evolves are:

$$\left.\begin{aligned} x_i^{00} &= x_{i\xrightarrow{0}}^{00} + x_{i\xrightarrow{0}}^{01} + x_{\Rightarrow i}^{00} \\ x_i^{01} &= x_{i\xrightarrow{0}}^{10} + x_{i\xrightarrow{1}}^{00} + x_{\Rightarrow i}^{01} \\ x_i^{10} &= x_{i\xrightarrow{0}}^{11} + x_{i\xrightarrow{1}}^{01} + x_{\Rightarrow i}^{10} \\ x_i^{11} &= x_{i\xrightarrow{1}}^{11} + x_{i\xrightarrow{1}}^{10} + x_{\Rightarrow i}^{11} \end{aligned}\right\} \quad (1)$$

$$\forall c \in \mathcal{C}, x_i^c = x_{i\xrightarrow{0}}^c + x_{i\xrightarrow{1}}^c + x_{i\Rightarrow}^c \quad (2)$$

The variables related to branch prediction are linked to the execution counts of basic blocks and edges by the following constraints:

$$x_i = \sum_c x_i^c \qquad \forall d \in \mathcal{D}, x_{i\xrightarrow{d}} = \sum_c x_{i\xrightarrow{d}}^c + \sum_c x_{i\overset{d}{\Rightarrow}}^c \quad (3)$$

For the initial and final state of the branch counter of block i, we can write:

$$\sum_c x_{\Rightarrow i}^c = 1 \qquad \sum_c x_{i\Rightarrow}^c = 1 \quad (4)$$

Finally, mispredictions counts are derived from:

$$m_i = x_{i\xrightarrow{1}}^{00} + x_{i\xrightarrow{1}}^{01} + x_{i\xrightarrow{0}}^{10} + x_{i\xrightarrow{0}}^{11} \quad (5)$$

This set of constraints has to be included for each basic block that ends with a conditional branch.

4 Branch prediction modeling and implicit path enumeration

4.1 Example code

We have analyzed the branch predictor behavior for our example code using the model described in the previous section. Figure 5 shows the results we obtained. The branch at the end of block 3 controls the inner loop that iterates 5 times (then the branch is executed 6 times for each execution of the loop: it is *not taken* 5 times and *taken* once). The inner loop is repeated 4 times: block 3 is then executed 24 times on the global execution path (20 times as *not taken* and 4 times as *taken*). The numbers given in the dotted and dashed hexagons stand for the correct and erroneous branch prediction counts.

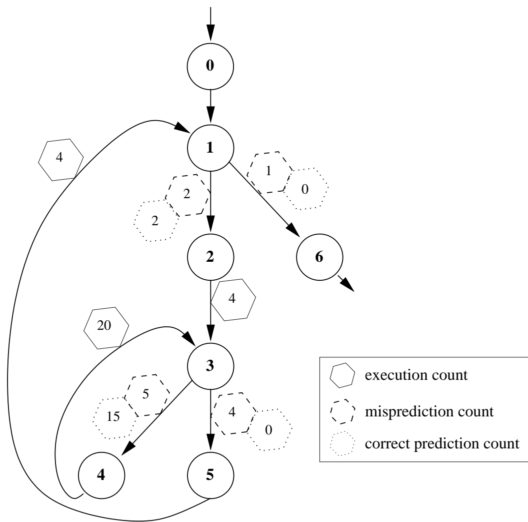

Figure 5. Results for the example code

In [1], the worst-case number of mispredictions for a branch that controls a repeated loop is bounded. For a loop that iterates N times ($N \geq 3$) and is repeated M times, the worst-case misprediction count is $(M + 2)$: during the first execution of the loop, the prediction counter reaches the `00` state after 3 iterations at most (considering any possible starting state) and the branch is mispredicted at most twice. At the end of every execution of the loop, the counter is incremented from `00` to `01` and the branch is mispredicted (this makes M mispredictions). For the next execution, the counter is decremented to `00` and the branch is well predicted.

According to these results, the worst-case misprediction count for the inner loop of our example should be 6. But the result obtained with the ILP model is 9, as shown in Figure 5. A closer look to these results show that they correspond to a behaviour of the prediction counter as the one shown in Figure 6 (a). This behaviour would be obtained if the branching pattern was:

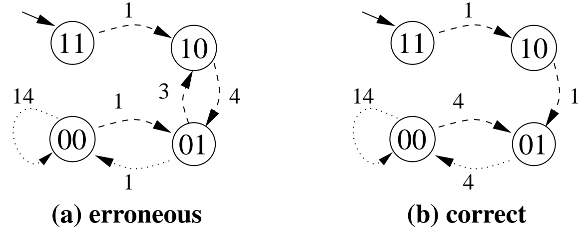

Figure 6. Detailed results for counter state transitions

$$\Big((\mathtt{NT}-\mathtt{NT}-\mathtt{T})-(\mathtt{NT}-\mathtt{T})_{\times 2}-(\mathtt{NT}_{\times 16}-\mathtt{T})\Big)$$

where `NT` stands for "*not taken*" and `T` for "*taken*". This pattern defines a path where the inner loop executes once with 2 iterations, then twice with one iteration, and finally once with 16 iterations: this path is inconsistent with the program semantics.

The only possible *explicit* path for this program has the same *implicit* description but has the following branching pattern:

$$(\mathtt{NT}-\mathtt{NT}-\mathtt{NT}-\mathtt{NT}-\mathtt{NT}-\mathtt{T})_{\times 4},$$

The correct evolution for the prediction counter is given shown in Figure 6 (b).

4.2 General case

In the general case (a loop with N iterations repeated M times), the number of mispredictions would be (over-) estimated as $(2M+1)$: the loop would considered as executing once with two iterations, then $(M - 2)$ times with a single iteration and finally once with $(M(N - 1))$ iterations. As said before, the correct value is $(M + 1)$.

The error comes from how flow information is expressed. The IPET formulation of the problem specifies that the edge that enters the inner loop is executed at most $M \times N$ times. This does not completely reflects the program semantics because the maximum number of iterations for each execution of the loop (*i.e.* N) is not specified. This missing parameter lets the ILP solver finding an infeasible path. Similar observations were mentioned for instruction cache analysis in [14]. However, the problem is more complex for branch prediction because it not possible to establish a direct link between paths in the CFG and transitions in the prediction counter finite-state automaton. In the next section, we show how additional constraints in the ILP model can control the branch predictor behavior for repeated loops.

5 Enforcing valid execution patterns for nested loops

5.1 Extended model

In [14], the worst-case miss rate for an instruction cache is evaluated by considering the possible states of cache lines. The state of a given cache line is the memory block it contains at some point of the program. The way this state changes is expressed by a Cache Conflict Graph (CCG). To some extent, a CCG plays the same role as the branch prediction counter automaton (shown in Figure 3). However, every edge in a CCG can be related to one path in the Control Flow Graph because each node of the CCG is related to a part of the code, *i.e.* to a node in the CFG. On the contrary, nodes in the branch prediction automaton do not stand for code parts and an edge in this automaton might correspond to several control flows.

To illustrate this, let us consider the transition from state `10` to state `01`. This transition can be fired either when the loop is entered (*i.e.* after block `b2` has been executed) or when it iterates (*i.e.* after `b4`). Then it is not possible to directly bound the execution count of this transition ($x^{10}_{i\xrightarrow{0}}$) to the execution counts of blocks `b2` and `b4` (this was the solution proposed in [14]).

In the case of a branch predictor, if a loop iterates at most N times, the prediction counter cannot fall into the `00` state more than N times it leaves this state. This guarantees that no more than N iterations are considered for each execution of the loop. This can be expressed by this additional constraint:

$$x^{01}_{i\xrightarrow{0}} + x^{00}_{i\xrightarrow{0}} + x^{00}_{\Rightarrow i} \leq N \times \left(x^{00}_{i\xrightarrow{1}} + x^{00}_{i\overset{1}{\Rightarrow}}\right)$$

This constraint applies to any loop with an upper-bounded number of iterations (which means that the effective number of iterations for one execution of the loop might range from 0 to N). This includes triangular loops where the number of executions of the inner loop depends on the value of the iteration counter of the outer loop.

This kind of constraint has to be generated for every block identified as controlling a loop. In the next section, we will give an overview of how the blocks that control loops can be identified from the CFG.

Considering the anomaly in the misprediction counts pointed out in this article, this constraint eliminates from the analysis some infeasible *explicit* paths related to valid *implicit* path. This is likely to tighten the WCET estimation.

5.2 Detecting loop-control blocks

As said before, integrating branch prediction into the IPET model makes it possible to consider various loop constructs. Our model fits different loop patterns (control at the beginning or at the end of the loop, exit of the loop either with a taken or not taken branch) as well as loops with multiple exits (however, to save room, we only describe the constraints for loops with a single exit).

This makes it necessary to identify in the CFG the blocks that contain a loop branch. Our algorithm implements predominance analysis to build sets of blocks that belong to a same loop. Then it searches, in each set, the block that has a successor out of the set: this block is the one that controls the loop and the direction of the branch to exit the loop is determined.

Once the block $\mathtt{b}_{\mathtt{ctrl}}$ that controls a loop has been identified, the number of executions of the loop is $x_{ctrl\xrightarrow{1}}$ (provided the loop is exited when the branch is taken).

5.3 Experimental results

We have made some experiments to measure the improvement due to refined branch prediction modeling. We have considered four benchmarks from the SNU suite [3]. The Control Flow Graphs of the programs were extracted and analyzed to identify the blocks that control loops using the OTAWA tool [7]. The block execution times were obtained using a cycle-level simulator that models a superscalar out-of-order processor and was developed in our team. Finally, the specification of the ILP problem for WCET calculation (*i.e.* the objective function and the structural and flow constraints) was produced by a perl script. We used `lp_solve` to solve the problem.

To estimate the impact of the infeasible paths on the calculated WCET, we have analyzed the benchmarks with both models: the earlier one, and the extended one proposed in this paper. Results are given in Table 1.

benchmark	**old WCET**	**new WCET**
`matmul`	3,246	3,078
`ludcmp`	11,411	11,201
`insertsort`	2,012	1,976
`crc`	211,628	210,098

Table 1. Estimated WCET.

It can be observed that the extended model gives tighter WCETs for all of the benchmarks we have tested. The improvement ranges from 0.72% to 5.17%. In every case, the earlier model over-estimates the number of branch mispredictions and then accounts for superfluous penalties.

6. Conclusion

Modeling advanced processor features using Integer Linear Programming and integrating the model to WCET estimation by IPET has several advantages: most of the loop

patterns can be analyzed, per-branch misprediction penalties can be accounted for, conditional structures are naturally analyzed. However, considering *implicit* paths is not sufficient to analyze schemes that behave according to the execution history. In the case of nested loops, cache miss or branch misprediction counts are overestimated because they are maximized for infeasible explicit paths. To get round this difficulty, we propose an extension to the branch predictor model. Experimental results show that the obtained WCET is tighter.

References

[1] I. Bate and R. Reutemann. Worst-Case Execution Time Analysis for Dynamic Branch Predictors. In *16th Euromicro Conference on Real-Time systems*, 2004.

[2] I. Bate and R. Reutemann. Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis. In *IEEE Conference on Real-Time Computing Systems and Applications*, 2005.

[3] SNU benchmark suite. http://archi.snu.ac.kr/realtime/benchmark/.

[4] C. Burguière and C. Rochange. A Contribution to Branch Prediction Modeling in WCET Analysis. In *Conference on Design, Automation and Test in Europe (DATE)*, 2005.

[5] C. Burguière and C. Rochange. Modélisation d'un prédicteur de branchement bimodal dans le calcul du WCET par la méthode IPET. In *13th International Conference on Real-Time Systems*, 2005.

[6] C. Burguière, C. Rochange, and P. Sainrat. A Case for Static Branch Prediction in Real-Time Systems. In *IEEE Conference on Real-Time Computing Systems and Applications*, 2005.

[7] H. Cassé and P. Sainrat. OTAWA, a Framework for Experimenting WCET Computations. In *3rd European Congress on Embedded Real-Time Software*, 2006.

[8] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for Processors with Branch Prediction. *Real-Time Systems*, 18(2-3), 2000.

[9] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor Support for Temporal Predictability - The SPEAR Design Example. In *Euromicro Conference on Real-Time Systems*, 2003.

[10] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards Industry-Strength Worst-Case Execution Time Analysis. Technical Report 99/02, ASTEC, 1999.

[11] C. Healy, R. Arnold, F. Muller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), 1999.

[12] X. Li, T. Mitra, and A. Roychoudhury. Modeling Control Speculation for Timing Analysis. *Real-Time Systems*, 29(1), 2005.

[13] Y.-T. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. *ACM SIGPLAN Notices*, 30(11), 1995.

[14] Y.-T. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *IEEE Real-Time Systems Symposium*, 1997.

[15] S.-S. Lim, S. Min, M. Lee, C. Park, H. Shin, and C. S. Kim. An Accurate Instruction Cache Analysis Technique for Real-Time Systems. In *Workshop on Architectures for Real-Time Applications*, 1994.

[16] T. Lundqvist and P. Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 17(2), 1999.

[17] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2), 1989.

[18] J. Smith. A Study of Branch Prediction Strategies. In *8th International Symposium on Computer Architecture*, 1982.