A Systematic Approach to Branch Speculation

TECHNICAL REPORT 97-25

Gianfranco Bilardi*
Università di Padova, Italy,
University of Illinois, Chicago

Alex Nicolau† Joe Hummel
University of California, Irvine

April 1997

Abstract

A general theoretical framework is developed for the study of branch speculation. The framework yields a systematic way to select the schedule in a given set that, for any (estimated) bias of the branch, minimizes the expected execution time. Among other things, it is shown that in some cases the optimal schedule is neither of those resulting from aggressively speculating on any given outcome of the conditional. Our results can be useful in either static or dynamic approaches. We propose a simple run-time estimator for the bias and discuss how to combine it with schedule selection. A number of examples motivate and illustrate the techniques, and show that our approach yields better performance in the case of highly unpredictable branches.

*DEI, Università di Padova, Padova, Italy, and EECS, University of Illinois, Chicago, IL 60607.
Supported in part by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM).

†This work supported in part by grants from ONR N00014-93-1-1348 and ARPA MDA904-96-C-1472.

A Systematic Approach to Branch Speculation

Gianfranco Bilardi*
Università di Padova, Italy,
University of Illinois, Chicago

Alex Nicolau† Joe Hummel
University of California, Irvine

April 1997

Abstract

A general theoretical framework is developed for the study of branch speculation. The framework yields a systematic way to select the schedule in a given set that, for any (estimated) bias of the branch, minimizes the expected execution time. Among other things, it is shown that in some cases the optimal schedule is neither of those resulting from aggressively speculating on any given outcome of the conditional. Our results can be useful in either static or dynamic approaches. We propose a simple run-time estimator for the bias and discuss how to combine it with schedule selection. A number of examples motivate and illustrate the techniques, and show that our approach yields better performance in the case of highly unpredictable branches.

1 Introduction

This paper investigates the relationship, interaction, and tradeoffs between branch prediction and speculative execution. In particular, we study the relationship between static (compile-time) and dynamic (run-time) branch prediction and speculation, and derive a rigorous frame-

*DEI, Università di Padova, Padova, Italy, and EECS, University of Illinois, Chicago, IL 60607. Supported in part by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM).

†This work supported in part by grants from ONR N00014-93-1-1348 and ARPA MDA904-96-C-1472. Please direct correspondence to this author at nicolau@ics.uci.edu.

work in which the interaction between these techniques can be understood. While the basic model used for illustrations throughout this paper is a VLIW model, the resulting framework is also useful for the understanding and future development of superscalar machines.

1.1 The machine model

The machine model we will use for our illustrations throughout this paper is a standard VLIW model, were each *instruction* can contain up to k operations. An operation can be an arithmetic or logical RISC type operation, a memory access (load/store) operation, or a conditional or unconditional jump. Each operation takes exactly one cycle to execute, and all operands in an instruction are read prior to assignment of any results computed during that instruction. Instructions are executed one at a time, with the *unique* next instruction to be executed being determined by the conditional(s) executed in the current instruction.

A *sequential schedule* for a given program consists of the RISC machine-level program control-flow graph of the program, with each node in the graph consisting of a single RISC operation. In our machine model, execution of a sequential schedule corresponds to completely sequential execution of the code.

A *compacted schedule* for a program consists of a transformed version of the sequential schedule, where some operations have been moved in order to execute in parallel with other operations. To be meaningful, the compacted schedule must preserve the semantics of the original schedule. Thus, for example, the move of an operation should not violate data-dependences. In effect, *compaction** attempts to fill a VLIW instruction with as many operations as possible—subject to data-dependences, control-flow and resource constraints—to shorten the execution time of the program.

1.2 Motivation

Control flow complicates the compaction process in two ways. Firstly, it imposes additional constraints on the motion of operations, e.g., an operation from the true branch of a conditional branch cannot be moved, as is, above the conditional if its destination is live (used) on the false branch.

*I.e., the process (technique) that derives a compacted schedule.

Secondly, and more important for the purpose of this paper, executing an operation from one branch of a conditional prior to the conditional gambles on the direction the conditional will take—in effect guessing that the operation should eventually be executed. If that branch is not taken, time was wasted not only because a useless operation was performed, but also because an opportunity to execute a useful operation (e.g. from the other branch) was missed. *Speculation* is the process of scheduling and executing operations before the direction a conditional will take is actually known.

Though not always a good idea, there are several reasons why speculation is quite useful. Most important is the fact that conditionals occur with surprisingly high frequency in most programs (one out of every 3-8 operations is a conditional jump, according to a variety of studies [NiFi84, Ku88]). These same studies indicate that to find substantial amounts of instruction-level parallelism one *has to* search for it across conditionals, since the amount of code between conditionals is too small to yield more than a factor of two parallelism, if that. Furthermore, some of the conditionals themselves will depend on the code that precedes them, thus delaying their execution and in a sense increasing the need for speculation.

In the presence of unlimited resources, the choice of which branch to speculate on would not be an issue, as we could guarantee that we execute all operations as early as possible (subject to data dependences, etc.) by simply aggressively speculating on both branches of every jump. Similarly, if we could predict, either dynamically or statically, that a branch of a conditional is always taken we could act accordingly. Indeed, there are situations where aggressive speculation on both branches is feasible, and one of the branches is virtually always taken. Trace Scheduling [Fi81] and Superblock [HMCC93] are based on the assumption that branches are very highly biased, and that this fact can be statically determined. Other techniques, like Percolation Scheduling [Ni85, Eb87] allow or even encourage aggressive speculation on both branches of a conditional. However, in the general case, the available limited resources combines with the presence of unpredictable (or dynamically predictable but unbiased) branches greatly complicate the issue of when and how to speculate. Dynamic schemes, implemented in virtually all superscalar processors available today, testify to the importance of such statically unpredictable, partially biased, or unbiased branches.

However, conventional wisdom holds that either branches are clearly biased towards one

branch (and then that branch should be chosen for speculation), or operations from both branches should be given equal priority when speculating. In particular, it is widely assumed that for branches that are not heavily biased, not much can be done at compile-time, beyond randomly selecting one or the other of the branches to be speculatively scheduled.

To our knowledge, the important issue of how best to speculate in the presence of such branches, and the related issue of how such static speculation compares with dynamic schemes, has not been investigated in detail.

1.3 The Problem

The problem actually consists of two separate issues: (a) how to predict/estimate branch bias, and (b) what to do with the result of the prediction in terms of scheduling.

We will show that as the probability of bias in a conditional changes from favoring the false branch to favoring the true branch, there are a sequence of (probability) sub-intervals that each correspond (have) a particular speculative schedule that is optimal for that interval. Of course, finding the optimal schedule for each interval is NP-H, and in practice heuristics will be used. However, given a set of heuristics that produce some schedules, our technique can be used to efficiently select the best schedules from within this set for each probability interval. Thus our approach could be used to improve the performance of a program in the presence of highly unpredictable, or even dynamically changing branching probabilities of conditional jumps.

In this paper we develop a framework for studying the above problems of speculation and branch prediction. We start by describing some examples that motivate our approach. We then introduce the framework upon which we base our analysis, and propose a new technique based on this framework. Next we present a prediction mechanism for branch bias (probability) suitable for integration with the proposed speculation technique. Finally, we present some real examples of applying our approach, followed by conclusions.

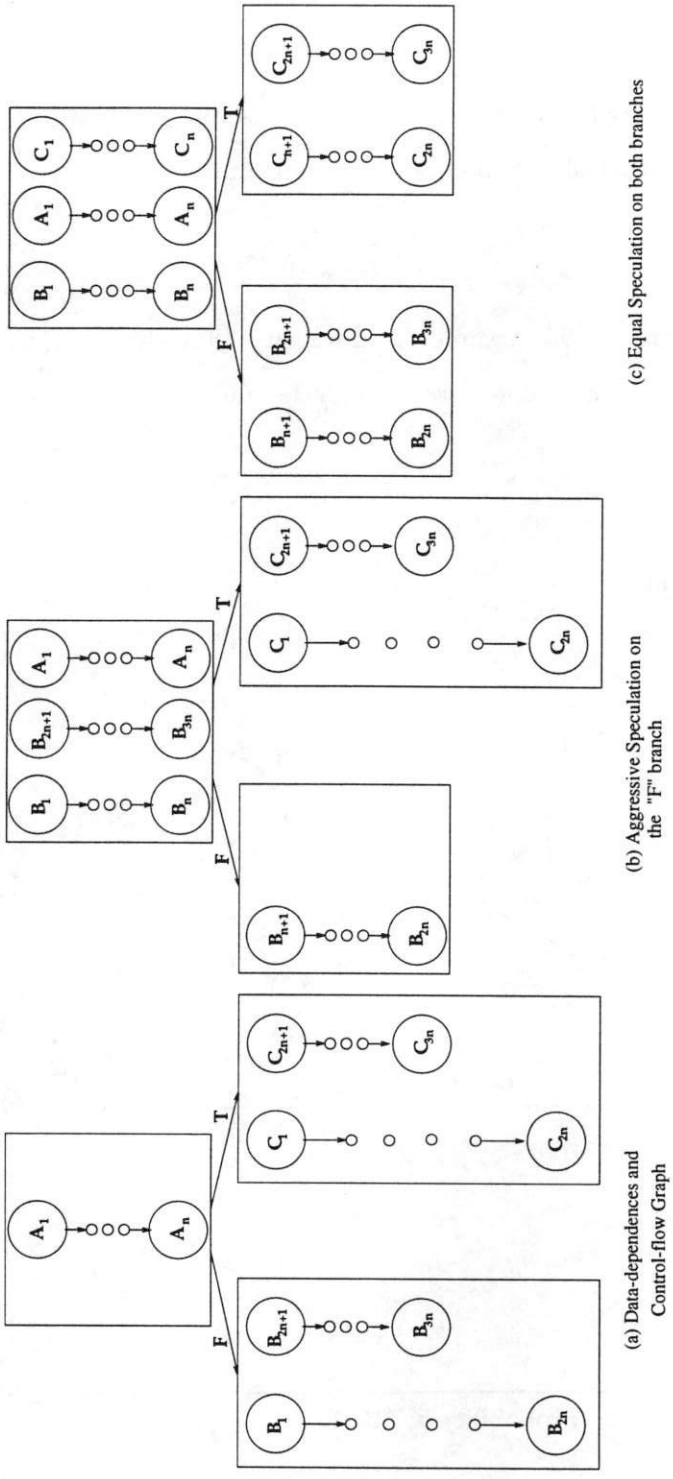

Figure 1: Failure of Aggressive One-branch Speculation (Example 1).

2 Some Examples

We present two theoretical examples, which serve as additional motivation. These examples illustrate that a schedule created through aggressive speculation on one branch is suboptimal for all probabilities of execution of the conditional statement.

2.1 Example 1

Consider the code in Figure 1. We assume for simplicity that the machine used for this example executes up to (any) three operations per cycle, and the operations have the data dependences shown in Figure 1a.

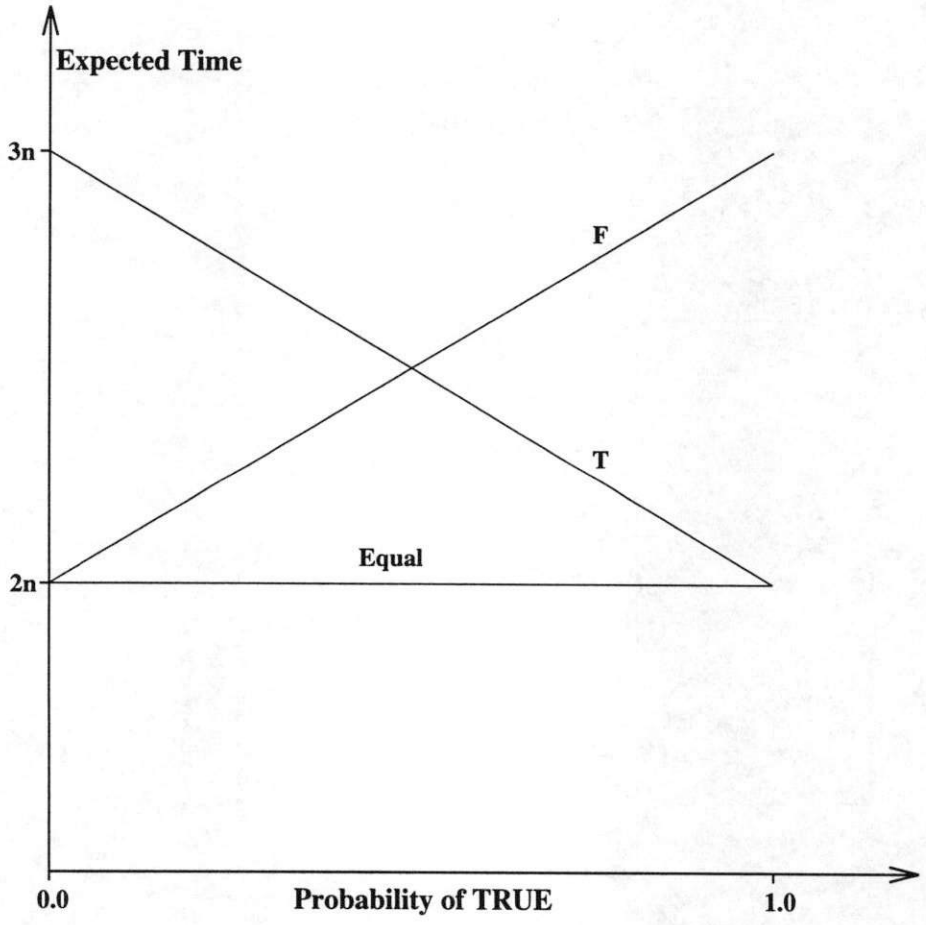

Figure 2: Time vs probability graph for Example 1.

Furthermore, the conditional statement is assumed to be operation A_n, i.e., A_1,...A_{n-1} have to execute (in order) before A_n can execute. Two possible compacted schedules are shown in Figures 1b and 1c. They correspond to aggressive speculation on the "F" branch (1b) and balanced speculation on both the "F" and "T" branch (1c). Aggressive speculation on the "T" branch is symmetric to that shown in Figure 1b and is thus omitted. Also, we omitted clearly suboptimal schedules such as speculating only on the non-critical paths of the branches.

Figure 2 relates branch probabilities and cycle counts. This figure shows that for all probabilities, the best schedule is obtained by aggressively scheduling across *both* branches. Existing static and dynamic approaches typically schedule only across one or the other branch (or switch back and forth between these schedules, incurring some penalty).

2.2 Example 2

Likewise, consider the code shown in Figure 3. This shows three different compacted schedules, all with differing cycle counts. Comparing probabilities versus cycle counts, we obtain the results shown in Figure 4. In this case, the best overall *schedule* again consists of all three compacted schedules, dynamically selected based upon probability. Note that as in Example 1, we have omitted suboptimal schedules from our presentation. We also do not show the the schedule that results from aggressive speculation across the "T" branch; it is symmetric to the "F" branch schedule shown in Figure 3b.

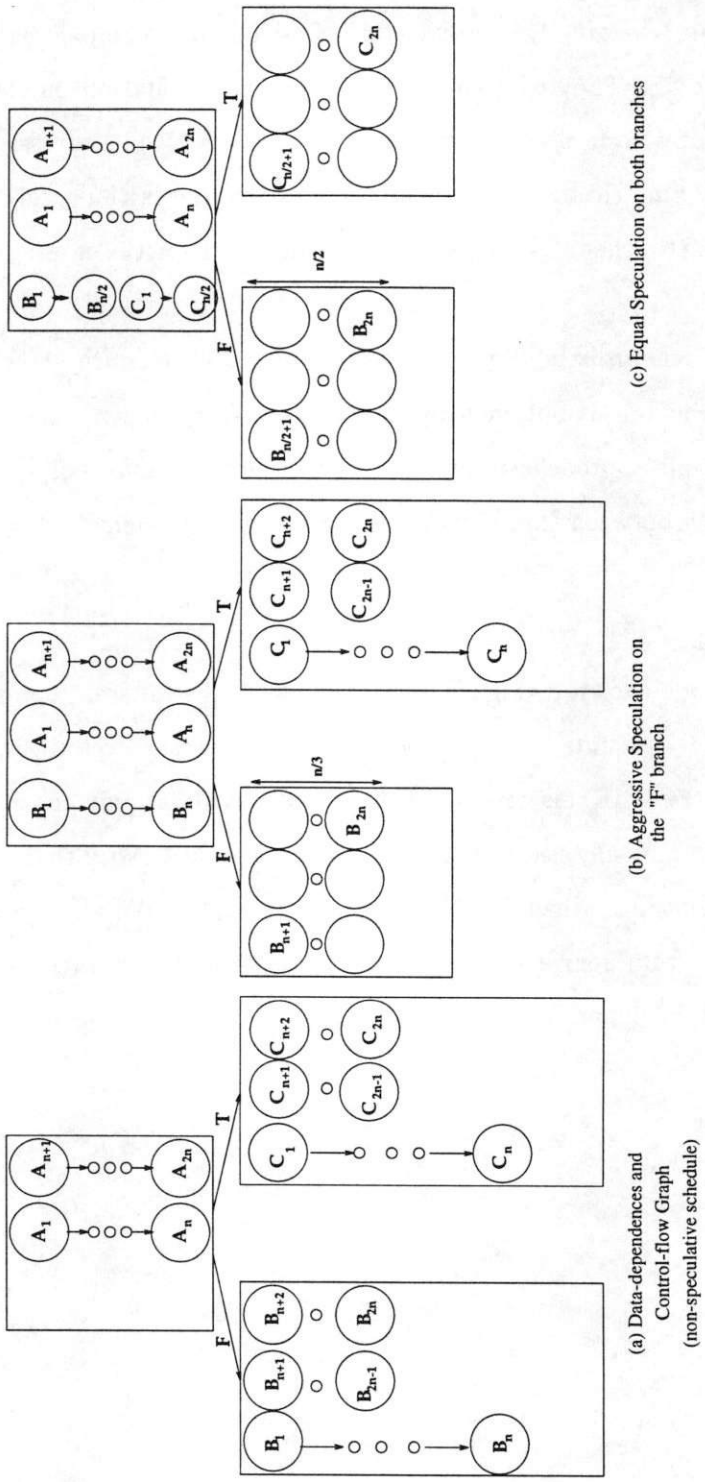

Figure 3: No SINGLE speculative approach yields optimal execution (Example 2).

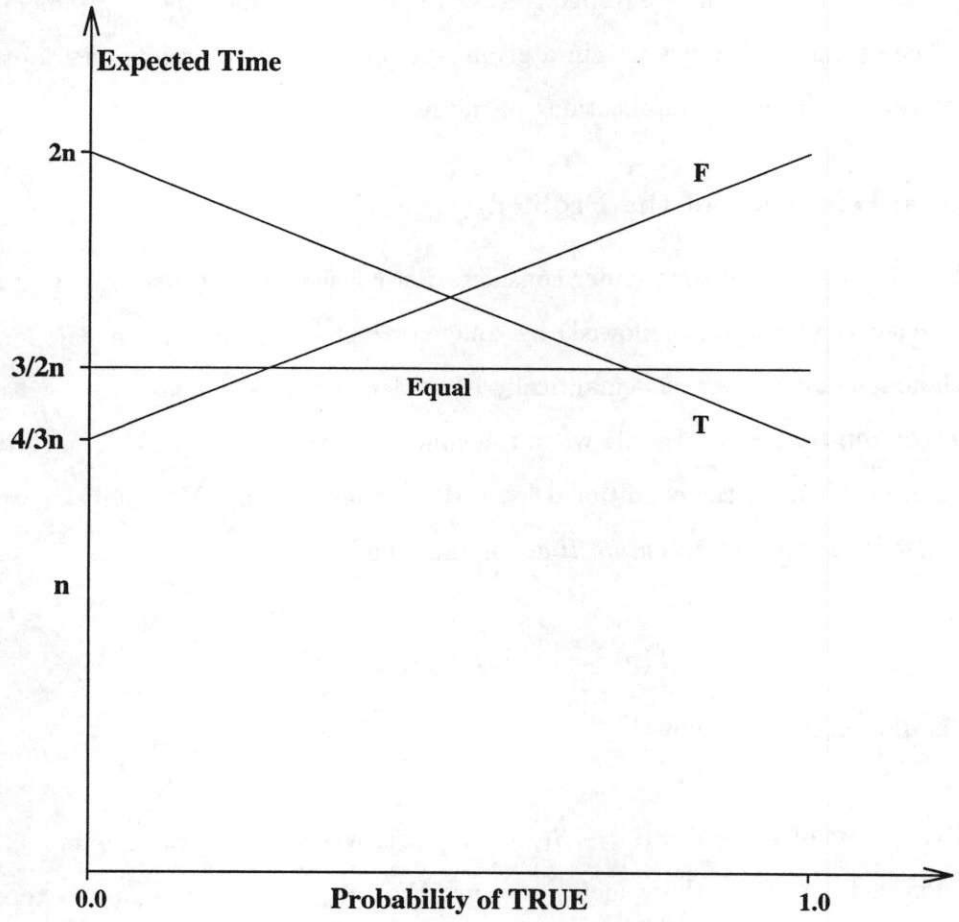

Figure 4: Time vs probability graph for Example 2.

3 The Framework

We now investigate systematically how to compare different schedules for a given segment of code containing one conditional statement. Assuming that somehow a set of schedules has been generated, the goal is to have a systematic procedure to select the best schedule as a function of the bias of the conditional.

The subsections of this sections are respectively devoted to (i) formalizing the problem, (ii) characterizing the optimal schedules within a given set, (iii) efficiently finding such optimal schedules, and (iv) discussing the implications of the results.

3.1 A Formal Definition of the Problem

We assume that the segment of code under consideration consists of a two-sided conditional statement, preceded (and possibly followed) by some straight-line code. A *schedule* for the code is a machine-language program semantically equivalent to it. We denote by f (respectively, t) the *execution time* of a schedule when the condition evaluates to false (respectively, true). We measure the *bias* of the conditional by p, the probability that the condition evaluates to true. Then, the *expected execution time* for the schedule is:

$$T(p) = (1-p)f + pt. \tag{1}$$

Our goal can be formulated as follows:

Problem: Given a set of schedules $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, where S_i has parameters f_i and t_i, determine, for each $p \in [0,1]$, the value(s) of i such that S_i yields the minimum expected execution time among the schedules in $\mathcal{S}$.

It is important to observe that, if $\mathcal{S}$ contains all possible schedules for the given code, then the solution of the above problem will yield the *absolutely optimal* schedules. On the other hand, if $\mathcal{S}$ is the result of a heuristic procedure to generate schedules, then the solution to the above problem will yield the schedules that are *relatively optimal* (i.e., the best within $\mathcal{S}$).

3.2 A Characterization of the Optimal Schedules

Since the expected time of a schedule S is uniquely determined by its parameters f and t, we find it convenient to work with the set of parameter pairs $A = \{(f_i, t_i) | S_i \in \mathcal{S}\}$ and to view A as a set of points in the plane. We say that a point $P_j = (f_j, t_j) \in A$ is *useless* within set A if, for any value of $p \in [0, 1]$, there is some $P_i = (f_i, t_i) \in A$ that yields equal or better expected time at p, i.e.:

$$T_i(p) = (1-p)f_i + pt_i \leq T_j(p) = (1-p)f_j + pt_j. \tag{2}$$

The points of A that are not useless will be called *useful.* The next theorem characterizes the set of useful points as well as the probability intervals of their usefulness. An example illustrating the theorem is shown graphically in Figure 5.

Theorem 1 *For any set A as above, the subset $A' \subseteq A$ of its useful points enjoys the following properties:*

1. *A' can be written as $\{P'_i = (f'_i, t'_i) | i = 1, \ldots, k\}$ so that the polygonal line π joining $P'_1, P'_2, \ldots, P'_k$ is the graph of an up-ward convex, decreasing function in the (f, t) plane, i.e.:*

   (a) *$f'_1 < f'_2 < \ldots < f'_k$ and $t'_1 > t'_2 > \ldots > t'_k$;*

   (b) *the quantity $s_i = (t'_i - t'_{i+1})/(f'_{i+1} - f'_i)$ decreases with i, for $i = 1, 2, \ldots, k-1$.*

2. *For any point $P = (f, t) \in A$, either (i) $f > f'_k$ and $t \geq t'_k$ or (ii) P lies on or above the polygonal line π.*

3. *Let $p_0 = 0$, $p_k = 1$, and $p_i = 1/(1 + s_i)$ for $i = 1, 2, \ldots, k-1$. Then, for $p \in [p_{i-1}, p_i]$, point (f'_i, t'_i) yields better expected time than any other point in A.*

Proof.

1. (a) Let us say that a point (f, t) is *strictly dominated* by a point (f', t') is $f \geq f'$, $t \geq t'$, and $(f, t) \neq (f', t')$. Clearly, if $(f', t') \in A$, then $(f, t) \notin A'$ since, according to Inequality (2), (f, t) is made useless by (f', t'). In fact,

$$T'(p) = (1-p)f' + pt' \leq T(p) = (1-p)f + pt,$$

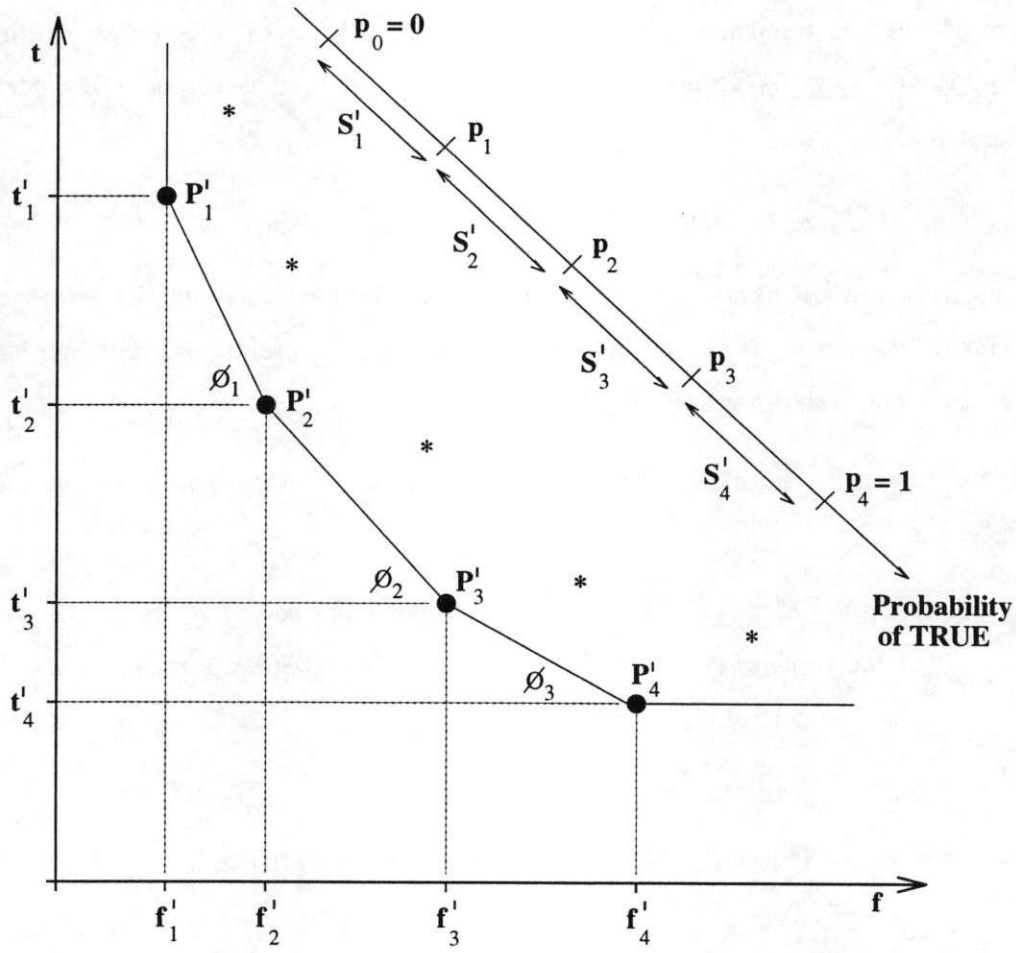

Figure 5: Example illustrating Theorem 1. Set A includes *useful* point P'_1, P'_2 P'_3, and P'_4 together with some *useless* points denoted by a "*". Schedules S'_1, S'_2 S'_3, and S'_4 respectively correspond to points P'_1, P'_2 P'_3, and P'_4. These schedules are respectively *optimal* within the probability intervals $[p_0 = 1, p_1]$, $[p_1, p_2]$, $[p_2, p_3]$, and $[p_3, p_4 = 1]$, where for $i = 1, 2, 3$, $p_i = (1 + \tan\phi_i)^{-1}$, with $\tan\phi_i = (t'_i - t'_{i+1})/(f'_{i+1} - f'_i)$. Observe how $\phi_1 > \phi_2 > \phi_3$ leads to $p_1 < p_2 < p_3$.

with equality holding at most for one value of p (i.e., $T'(0) = T(0)$ if $f = f'$ or $T'(1) = T(1)$ if $t = t'$).

Then, if we sort the point in A' by increasing abscissa, $f'_1 < f'_2 < \ldots < f'_k$, they will be also sorted by decreasing ordinate $t'_1 > t'_2 > \ldots > t'_k$. Otherwise, if for some i we had $t'_{i+1} \geq t'_i$, point (f'_{i+1}, t'_{i+1}) would be strictly dominated by point (f'_i, t'_i).

(b) By contradiction suppose that, for some i, $s_{i-1} \geq s_i$, that is,

$$(t'_{i-1} - t'_i)/(f'_i - f'_{i-1}) \geq (t'_i - t'_{i+1})/(f'_{i+1} - f'_i). \tag{3}$$

Since $f'_{i-1} < f'_i < f'_{i+1}$, for some $\alpha \in (0,1)$, we can write

$$f'_i = (1-\alpha)f'_{i-1} + \alpha f'_{i+1}. \tag{4}$$

By using Equation (4) in Inequality (3), after straightforward manipulations we obtain:

$$t'_i \geq (1-\alpha)t'_{i-1} + \alpha t'_{i+1}. \tag{5}$$

Relations (3) and (4) and some simple algebra yield:

$$\begin{aligned} T'_i(p) &= (1-p)f'_i + pt'_i \\ &\geq (1-p)[(1-\alpha)f'_{i-1} + \alpha f'_{i+1}] + p[(1-\alpha)t'_{i-1} + \alpha t'_{i+1}] \\ &= (1-\alpha)T'_{i-1}(p) + \alpha T'_{i+1}(p) \\ &\geq \min(T'_{i-1}(p), T'_{i+1}(p)), \end{aligned}$$

with equality in the last line holding only for the unique value of p such that $T'_{i-1}(p) = T'_{i+1}(p)$.

In conclusion, the expected time for point P'_i is always equal or worse than either that one for point P'_{i-1} or that one for point P'_{i+1}. Therefore, $P'_i \notin A'$, a contradiction.

2. A simple inductive argument will show that if a point $(f,t) \in A$ is useless, then for any p,

$$T(p) = (1-p)f + pt \geq \min_{i=1}^{k} T'_i(p). \tag{6}$$

It follows that $f \geq f'_1$; otherwise, $T(0) = f < f'_i = T'_i(0)$ for any i and Inequality (6) would be violated at $p = 0$.

It also follows that $t \geq t'_k$; otherwise $T(1) = t < f'_i = T'_i(1)$ for any i and Inequality (6) would be violated at $p = 1$.

Let us then assume that $f'_1 \leq f \leq f'_k$ and, more specifically, that $f'_{i-1} \leq f \leq f'_i$. Then, for some $\alpha \in (0, 1)$, we can write

$$f = (1 - \alpha)f'_{i-1} + \alpha f'_i. \tag{7}$$

If, by contradiction, (f, t) lies below polygonal line π, and in particular below the segment from (f'_{i-1}, t'_{i-1}) to (f'_i, t'_i), then

$$t < (1 - \alpha)t'_{i-1} + \alpha t'_i. \tag{8}$$

By developments similar to those of part 1(b) of the present proof, we can use Relations (7) and (8) to derive

$$T(p) < (1 - \alpha)T'_{i-1}(p) + \alpha T'_i(p). \tag{9}$$

It is easy to verify that, for $p = p_i = 1/(1 + s_i)$, with $s_i = (t_i - -t_{i+1})/(f_{i+1} - f_i)$, we have $T'_i(p_i) = T'_{i-1}(p_i)$. Hence, Inequality (9) yields:

$$T(p_i) < (1 - \alpha)T'_i(p_i) + \alpha T'_i(p_i) = T'_i(p_i) = T'_{i-1}(p_i). \tag{10}$$

The latter relation shows that, at $p = p_i$, point (f, t) yields a better expected time than points $P'_{i-1} = (f'_{i-1}, t'_{i-1})$ and $P'_i = (f'_i, t'_i)$. Since, according to part 3 of the theorem statement (proven below), for $p = p_i$, points P'_{i-1} and P'_i yield the best expected time of all points in A, we reach a contradiction. In conclusion, $p = (f, t)$ lies on ar above line π.

3. From the very definition of set A', it follows that, for any given value of $p \in [0, 1]$, the minimum expected time achievable by any schedule in $\mathcal{S}$ is

$$T_{min}(p) = \min_{i=1}^{k} T'_i(p). \tag{11}$$

It is readily verified that p_i (as defined in the theorem statement) is the value for which $T'_i(p_i) = T'_{i+1}(p_i)$. Since $p_i = 1/(1 + s_i)$, from part 1(b), we observe that p_i increases

with i. If we focus on the interval $[p_{i-1}, p_i]$ and consider that the graph of $T'_j(p)$ is a straight line, we can make the following observations:

(i) For $j < i$, the (graph of) $T'_j(p)$ starts lower than $T'_i(p)$ (since $T'_j(0) = f'_j < f'_i = T'_i(0)$) and intersects it at the left of p_{i-1}. Hence, for $p \geq p_{i-1}$, $T'_i(p) \leq T'_j(p)$.

(ii) For $j > i$, the (graph of) $T'_j(p)$ ends lower than $T'_i(p)$ (since $T'_j(1) = t'_j < t'_i = T'_i(1)$) and intersects it at the right of p_i. Hence, for $p \leq p_i$, $T'_i(p) \leq T'_j(p)$.

Therefore, for $p \in [p_{i-1}, p_i]$, $T'_i(p) \leq T'_j(p)$, for any j.

In conclusion, Equation (11) can be rewritten as

$$T_{min}(p) = T'_i(p) \quad \text{for} \quad p \in [p_{i-1}, p_i], \tag{12}$$

completing the proof of part 3.

3.3 An Algorithm to Filter out the Useful Schedules

Next, we develop an efficient algorithm to compute the useful points A' of a given set A. The key steps of the algorithm are as follows:

1. Input A.

2. Sort A by non decreasing abscissa (f) and, in case of tie, by increasing ordinate (t). Call the resulting sequence of points A_1.

3. Scan sequence A_1 producing a subsequence A_2 as follows. Let $Q = (f, t)$ be the point being currently scanned in A_1. Let $Q' = (f', t')$ be the last point of A_1 inserted in A_2. Initialize Q, Q' and A_2 to the first element in A_1. For each $Q \in A_1$, if $t < t'$ then append Q to A_2 and let $Q' = Q$.

4. Scan sequence A_2 producing a subsequence A_3, as follows. Denote by Q_1 and Q_2 the first and the second point in A_2, respectively. Then, execute the following procedure:

append Q_1 and Q_2 to (an initially empty) A_3;
$Q_a := Q_1$;
$Q_b := Q_2$;
for $Q_c \in A_2 - \{Q_1, Q_2\}$ **do**
begin
append Q_c to A_3;
while $Q_a \neq$ *null* **and** slope$(Q_bQ_c) \leq$ slope(Q_aQ_b) **do**
begin
remove Q_b from A_3;
$Q_b := Q_a$;
$Q_a :=$ predecessor$_{A_3}(Q_b)$ (null if Q_b is the first element of A_3)
endwhile
$Q_a := Q_b$;
$Q_b := Q_c$;
endfor

If we let $Q_x = (f_x, t_x)$, for $x = a, b, c$, then slope$(Q_bQ_c) = (t_c - t_b)/(f_c - f_b)$ and slope$(Q_aQ_b) = (t_b - t_a)/(f_b - f_a)$ so that the condition of the while statement can be rewritten as $(t_c - t_b)(f_b - f_a) \leq (t_b - t_a)(f_c - f_b)$.

5. Output $A' = A_3$.

The correctness of the algorithm is established by observing that Step 2 discards all points of A that are dominated by some other point in A, and Step 3 discards all points that lie on or above the segment that joins any two other points. In essence, Step 3 is a straightforward adaptation of Graham's algorithm [Gra72] for finding the convex hull of a planar set. Indeed, it follows from Theorem 1 that the convex hull of set $A \cup \{(0, \infty), (\infty, 0)\}$ is exactly $A' \cup \{(0, \infty), (\infty, 0)\}$.

The running time of the algorithm is $O(n \log n)$ for the sorting in Step 1, and $O(n)$ for each of Step 2 and Step 3. In Step 2, a constant amount of time is spent on each element of list A_1, and $|A_1| \leq n$. Constant work per element is also done in Step 3; in fact, each iteration of the for (respectively, while) loop appends to (respectively, removes from) A_3 a

different element of A_2, and $|A_2| \leq n$.

In conclusion, we have:

Theorem 2 *Set A' can be obtained from set A in time $O(n \log n)$.*

3.4 On Schedule Generation

The previous sections show how to select the best schedules out of a given set. Of course, the question remains of how to generate a set of schedules guaranteed to obtain the optimal ones. The problem is known to be NP-hard in the presence of limited resources, even for straight-line code. Nevertheless, Theorem 1 is helpful in formulating the specific type of scheduling problem that arises in our context, and sheds some light on possible heuristics.

Let $\mathcal{S}$ be the set of *all* possible schedules for a given code. Let $f(S)$ (respectively, $t(S)$) be the time to execute schedule $S \in \mathcal{S}$ when the condition is false (respectively, true). We are interested in the following integer function τ of the integer variable f, with the convention that $\min \emptyset = +\infty$:

$$\tau(f) = \min\{t(S) : f(S) \leq f, S \in \mathcal{S}\}. \tag{13}$$

In general, $\tau(f)$ is a non increasing function with $\tau(f) = +\infty$ for $f < f'_1$ and that $\tau(f) = \tau(f'_k) = t'_k$ for $f > f'_k$ (f'_1, f'_k, and t'_k as in Theorem 1).

It is easy to verify that no optimal schedule is discarded in the following subset of $\mathcal{S}$:

$$\mathcal{S}_\tau = \{S \in \mathcal{S} : t(S) = \tau(f(S)), \;\; f'_1 \leq f(S) \leq f'_k\}.$$

Then, the optimal schedules can be identified by applying our techniques to the set $A_\tau = \{((f(S), t(S)) : S \in \mathcal{S}_\tau\}$ and computing the corresponding A'_τ. In general, A'_τ may be a proper subset of A_τ, since when A_τ is arranged by increasing abscissa, the corresponding sequence of ordinates is non increasing, but not necessarily strictly decreasing or up-ward convex.

We outline a strategy to find the optimal solutions:

1. Compute $f'_1 = \min\{f(S) : S \in \mathcal{S}\}$.

2. Compute $t'_k = \min\{t(S) : S \in \mathcal{S}\}$.

3. Compute $f_k = \min\{f(S) : t(S) \leq t'_k,\ S \in \mathcal{S}\}$.

4. Compute $\tau(f) = \min\{t(S) : f(S) \leq f,\ S \in \mathcal{S}\}$, for $f'_1 \leq f \leq f'_k$, and the corresponding $\mathcal{A}_\tau$

5. Compute A'_τ from A_τ by Theorem 2.

Steps 1 and 2 of the outlined strategy require the solution of a minimum schedule problem for a straight-line code. Steps 3 and 4 require the solution of the more general problem where the length of the schedule for one-branch is minimized, subject to an upper bound on the length of the schedule for the complementary branch.

As the above scheduling problems are NP-hard, one may have to settle for some heuristics and hence obtain only an approximate answer in Steps 1 to 4. However, it is important to observe that Step 5 will always yield the best schedules among those produced by the heuristics.

3.5 The Restructured Code

Once a set of (absolutely or relatively) optimal schedules $S_1, S_2, \ldots, S_k$ are obtained, with the corresponding probability intervals as in Theorem 1, the original code can be replaced by the a semantically equivalent code structured as follows:

1. input(p);

2. find (smallest) i such that $p_{i-1} \leq p \leq p_i$;

3. execute code for schedule S_i;

The value of p could be decided statically, from analysis of original code structure or from results of suitable execution profiles; then, Steps 1 and 2 would actually be compiler's tasks and the target code would reduces to just the code for S_i.

Alternatively, p could be estimated at run time, by either software or hardware techniques; then, each of the k schedules must be in the target code.

4 Estimating the Bias at Run Time

Given a code segment with the properties assumed in the previous section, let C denote the condition and, for $h = 1, 2, \ldots$, let $c_h \in \{0 = false,\ 1 = true\}$ be the value taken by condition C at the h-th execution of the conditional statement (when the program runs on some fixed input data).

We view the (Boolean) sequence $c_1, c_2, \ldots$ as a random process. The goal is to estimate the conditional probability $r_h = Pr(c_h = 1 | c_1, c_2, \ldots, c_{h-1})$. Arbitrarily sophisticated schemes could be adopted to perform this estimate, and any of them could be coupled with the strategy developed in the previous section. Here, we shall propose a simple and effective estimator in order to demonstrate that the bias estimate can be obtained with negligible (hardware or software) overhead.

Our estimator is a first-order filter, with an adjustable parameter $\beta \in [0, 1]$. Denoting by b_h the estimate of r_h, the estimator is described the the following recurrence relation:

$$b_{h+1} = (1 - \beta) b_h + \beta c_h \ , \qquad h > 1, \tag{14}$$

with initial condition b_1, to be statically selected. It is interesting to observe that the proposed estimator is completely symmetric with respect to false and true branches. In fact, if we let $q_h = (1 - b_h)$ be the estimate for the bias of the false branch, we can easily rewrite Equation (14) as

$$q_{h+1} = (1 - \beta) q_h + \beta (1 - c_h) \ , \qquad h > 1, \tag{15}$$

with $q_1 = 1 - b_1$, which has the same form as Equation (14) does.

Below are a few, simple to derive, properties of the estimator, which help to develop some intuition on its behavior.

- If $0 \leq b_1 \leq 1$, then for every $h \geq 1$, we have $0 \leq b_h \leq 1$.
- If for every h we have $c_h = 0$, then $b_h = (1 - \beta)^h b_1$, which converges to 0 exponentially, the larger β the faster. (Symmetrically, if $c_h = 1$, b_h converges to 1.)
- If $\beta = 0$, then $b_h = b_1$, for any h. In this case, there is no dynamic update of the estimate, which remains uniquely determined by the statically selected initial condition.

- If $\beta = 1$, then $b_{h+1} = c_h$, for any h. In this case, the condition is estimated to be completely biased according to the previous outcome of the condition (except for the first execution).

- In general, $|b_{h+1} - b_h| \leq \beta$. As β increases from 0 to 1, the resulting estimator becomes more sensitive to the most recent history of the condition and less sensitive to the remote history. Hence, for larger values of β, the estimator responds more quickly to changes in bias, but is correspondently less stable.

From the standpoint of implementation, we observe that to perform one update of the bias estimate only requires a small number of operations (see Equation (14)). These operations can be further simplified by choosing $\beta = 2^{-j}$. In this case, Equation (14) can be rewritten as

$$b_{h+1} = b_h - 2^{-j} b_h + 2^{-j} c_h \ , \quad h > 1, \tag{16}$$

only requiring one shift, one subtraction, and one addition (if $c_h = 1$).

4.1 Integrating the Estimator with the Schedules

We recall that, when we use the approach developed in the preceding section, the probability interval $[0, 1]$ is partitioned into subintervals $[p_{i-1}, p_i]$, for $i = 1, 2, \ldots, k$, and to each interval there correspond an optimal schedule to be used when p is in that interval. Of course, as p is not known, we will work with its estimate b instead.

Let i_h be such that $b_h \in [p_{i_h - 1}, p_{i_h}]$. It is interesting to observe that, since $|b_{h+1} - b_h| \leq \beta$, if $\beta < \max_j (p_j - p_{j-1})$, then $|i_{h+1} - i_h| \leq 1$. More specifically, we have:

if $c_h = 0$ **then**
  if $b_{h+1} \geq p_{i_h - 1}$
    then $i_{h+1} = i_h$
    else $i_{h+1} = i_h - 1$
else
  if $b_{h+1} \geq p_{i_h}$

then $i_{h+1} = i_h + 1$

else $i_{h+1} = i_h$

(If variable I holds the current value of i_h, then the above code can be further simplified). Between consecutive executions of the condition, control moves only between consecutive schedules in the sequence.

5 A Real Example

Consider the following C code fragment, taken from a multimedia application (one of the applications used to motivate the MMX instruction set [PWW97]):

```
for (i = 1; i <= N; i++)
  if (a[i] == K)
    out[i] = c[i];
  else
    out[i] = b[i];
```

Assuming that each VLIW instruction may contain up to 3 operations (of any kind per cycle), the loop compiles into the following VLIW instruction stream S1:

```
        i  = 0          | N' = N * 4
Loop:   r1 = a + i
        r2 = Mem[r1]
        jneq r2,K,Else
Then:   r3 = out + i    | r4 = c + i     | i  = i + 4
        r5 = Mem[r4]
        Mem[r3] = r5    | jlt  i,N',Loop | jge  i,N',Done
Else:   r3 = out + i    | r4 = b + i     | i  = i + 4
        r5 = Mem[r4]
        Mem[r3] = r5    | jlt  i,N',Loop
Done:
```

This code is optimized to maximize parallelism *without* scheduling across either branch. Note that variables are referenced by name for readability purposes; these variables would occupy registers in the final schedule. Each iteration of the loop requires 6 cycles.

Now, suppose we assume the conditional is always TRUE. Scheduling aggressively across the then-branch yields the following schedule S2:

```
        i  = 0          | N' = N * 4
Loop:   r1 = a + i      | r3 = out + i   | r4 = c + i
        r2 = Mem[r1]    | r5 = Mem[r4]   | i = i + 4
        jneq r2,K,Else  | Mem[r3] = r5   | jlt i,N',Loop
        jge  i,N',Done
Else:   r4 = b + 1
        r5 = Mem[r4]
        Mem[r3] = r5    | jlt i,N',Loop
Done:
```

In this case, the operations from the then-branch are moved up to fill the resources available in the instructions that perform the conditional test. As a result, each iteration of the loop now requires only 3 cycles (4 on the last iteration) if the conditional is in fact TRUE, and 6 cycles if the conditional turns out to be FALSE.

If we assume the opposite, that the conditional is always FALSE, the results are symmetric: each iteration of the loop requires only 3 cycles (4 on the last iteration) if the conditional is in fact FALSE, and 6 cycles if it turns out to be TRUE. We denote this schedule symmetric S2.

Finally, suppose we assume that the branches are equally likely, i.e. that the conditional could be TRUE or FALSE with equal probability. Scheduling aggressively across *both* branches yields the schedule S3:

```
        i  = 0          | N' = N * 4
Loop:   r1 = a + i      | r3 = out + i   | r4 = c + i
        r2 = Mem[r1]    | r5 = Mem[r4]   | r4' = b + i
        jneq r2,K,Else  | r5' = Mem[r4'] | i = i + 4
Then:   Mem[r3] = r5    | jlt i,N',Loop  | jge i,N',Done
Else:   Mem[r3] = r5'   | jlt i,N',Loop
Done:
```

In this case, the available resources are split between the needs of both branches, resulting in a schedule that requires 4 cycles for each iteration of the loop—regardless of the value of the conditional.

Table 1 compares the cycle times for 1000 iterations of the loop, under varying branch probabilities and execution strategies. Each row denotes the probability of one branch executing in favor of the other. The first four rows reflect a random distribution of branches. The latter rows all denote probabilities of 0.5, but where the branch distribution is based on

an alternating sequence of T and F branches: d denotes the *duration* (number of iterations) before the branch switches from T to F (or F back to T). The columns denote different strategies for executing the loop, the first three denoting compile-time approaches and the fourth denoting a typical run-time approach. It is assumed that in the run-time case, the penalty for switching between schedules is 0 cycles (this is very optimistic). On the other hand, the run-time approach is using a two-bit predictor, and thus takes two iterations to recognize a sustained switch in the control flow of the conditional. Our approach is using a four-bit history estimator.

N=1000	static (S2)		static (avg S3)	**our approach (speculation S2-S3)**	dynamic (HW speculation S1)
	picked correct	picked wrong			
p = 1	3000	6000	4000	**3000**	3000
p = 0.9	3312	5685	4000	**3354**	3365
p = 0.7	3886	5111	4000	**3993**	4133
p = 0.5	4522	4477	4000	**4287**	4455
p = 0.5 (d=1)	4500	4500	4000	**4001**	4500
p = 0.5 (d=2)	4500	4500	4000	**4000**	5994
p = 0.5 (d=3)	4500	4500	4000	**3999**	4995
p = 0.5 (d=4)	4500	4500	4000	**4494**	4494
p = 0.5 (d=5)	4500	4500	4000	**4194**	4194

Table 1: Cycles required to execute multimedia loop.

Statically scheduling across one branch enables a 3-cycle iteration if execution follows the scheduled branch, but 6 cycles if not. As shown in Table 1, this does well when the probability is high AND the compiler happens to pick the correct branch. Statically scheduling for the best average case does well for lower probabilities, but is unable to exploit high probabilities of one branch being taken over the other. The dynamic approach uses hardware-based branch prediction at run-time to generate a schedule, either S2 or symmetric S2. This performs well when the flow of execution is highly-predictable, and when the branches do not alternate frequently between T and F.

Our approach, which generates 3 schedules at compile-time (S2, symmetric S2, and S3) AND then selects one during each iteration at run-time, performs well across a range of probabilities. In particular, our approach always performs as well OR better than the dynamic

scheme. In comparison to static schemes, our approach generally does better, though one can construct cases in which our technique does worse (e.g. p=0.5 with d=4). This is a function of the estimator's accuracy, in particular how much history it can retain. For example, Table 1 shows that the worst-case cycle time for the dynamic case is 5994 (p=0.5,**d=2**), which is due to its use of a two-bit predictor. In our approach, the worst-case cycle time is 4494 (p=0.5,**d=4**), given our four-bit estimator.

5.1 Additional examples from other Applications

Please note: the final version of the paper will contain additional examples taken from other benchmarks. Regardless, we feel that the point of the paper has been made, given the strong theoretical foundation and the demonstration of its potential on a real example.

6 Conclusions

We presented a general theoretical framework for the study of branch speculation. The framework yields a systematic way to select the schedule in a given set that, for any (estimated) bias of the branch, from aggressively speculating on any given outcome of the conditional. We showed that in some cases, the optimal schedule is a combination of schedules, and not those resulting only from aggressively speculating on any given outcome of the conditional. This approach yields better performance than existing static and dynamic techniques in the case of highly unpredictable branches, and performs no worse otherwise. This work is useful for both static and dynamic approaches to branch speculation.

References

[Eb87] K.Ebcioglu. "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps". *Proceedings of the 20th Annual Workshop on Microprogramming*, pp. 69-79, ACM Press, 1987.

[Fi81] J. A. Fisher. "Trace Scheduling: A technique for global microcode compaction". *IEEE Transactions on Computers*, No. 7,pp. 478-490, 1981.

[Gra72] R. Graham. An efficient algorithms for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[HMCC93] W. Hwu, S. Mahlke, W. Chen, and P. Chang. The Superblock – an effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, May 1993.

[Ku88] M. Kumar. "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications". *IEEE Trans. on Computers*, Vol 37, No. 9, pp. 1088-1098, September 1988.

[Ni85] A. Nicolau. "Uniform Parallelism Exploitation in Ordinary Programs". *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

[NiFi84] A. Nicolau, J. Fisher. "Measuring the parallelism available for VLIW architectures". *IEEE Trans. on Computers*, C-33, pp. 968-976, November 1984.

[PWW97] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):25–38, January 1997.