# Difficult-Path Branch Prediction Using Subordinate Microthreads

Robert S. Chappell† Francis Tseng‡ Adi Yoaz# Yale N. Patt‡

†EECS Department
The University of Michigan
Ann Arbor, Michigan 48109-2122
robc@eecs.umich.edu

‡ECE Department
The University of Texas at Austin
Austin, Texas 78712-1084
{tsengf, patt}@ece.utexas.edu

# Texas Development Center
Intel Corporation
Austin, TX 78746
adi.yoaz@intel.com

## Abstract

*Branch misprediction penalties continue to increase as microprocessor cores become wider and deeper. Thus, improving branch prediction accuracy remains an important challenge. Simultaneous Subordinate Microthreading (SSMT) provides a means to improve branch prediction accuracy. SSMT machines run multiple, concurrent microthreads in support of the primary thread. We propose to dynamically construct microthreads that can speculatively and accurately pre-compute branch outcomes along frequently mispredicted paths. The mechanism is intended to be implemented entirely in hardware. We present the details for doing so. We show how to select the right paths, how to generate accurate predictions, and how to get this information in a timely way. We achieve an average gain of 8.4% (42% maximum) over a very aggressive baseline machine on the SPECint95 and SPECint2000 benchmark suites.*

## 1. Introduction

Branch mispredictions continue to be a major limitation on microprocessor performance, and will be for the foreseeable future [9]. Though modern prediction mechanisms can achieve impressive accuracies, the penalty incurred by mispredictions continues to increase with wider and deeper machines. For example, a futuristic 16-wide, deeply-pipelined machine with 95% branch prediction accuracy can achieve a twofold improvement in performance solely by eliminating the remaining mispredictions[1].

Simultaneous Subordinate Microthreading [2] has the potential to improve branch prediction accuracy. In this approach, the machine "spawns" subordinate *microthreads* to generate some of the predictions, which are used in place of the hardware predictions. Because microthreads are not limited by hardware implementation and can target very specific behavior, they can generate very accurate predictions. Examples have been shown in previous research [2, 18].

However, effective use of microthreads for branch prediction is challenging. Microthreads compete with the primary thread for resources, potentially decreasing performance. Because of this drawback, subordinate microthreading is not a good general solution—it must be targeted such that mispredictions are removed without incurring so much overhead that the performance gains are overshadowed. To maximize gains, it is important to have the following goals:

- **Spawn only useful threads.** A microthread incurs useless overhead if it generates a prediction for a correctly predicted branch, or if a microthread is spawned but the prediction is never used. Additionally, if the branch is correctly predicted, there is a risk of introducing additional mispredictions with microthreads.
- **Generate accurate microthread predictions.** A microthread is useless if it does not generate a correct prediction. In fact, it can be harmful if the hardware prediction is correct and the microthread prediction is not.
- **Complete microthreads in time to be useful.** A microthread is useless if it does not complete before the target branch is resolved—at that point, the machine has already computed the actual outcome of the branch. Microthreads should complete as early as possible to eliminate or reduce misprediction penalties.

We propose using a set of *difficult-paths* to guide microthread branch prediction with the above goals in mind. Path-based confidence mechanisms [10] have demonstrated

[1] Averages over SPECint95 and SPECint2000. See Section 5 for experimental setup.

1063-6897/02 $17.00 © 2002 IEEE

that the predictability of a branch is correlated to the control-flow path leading up to it. We extend this notion by classifying the predictability of branches by control-flow path, rather than as an aggregate of all paths to a branch. In this manner, we identify *difficult paths* that frequently lead to mispredictions. These are the mispredictions we attack with specially-constructed microthreads.

The remainder of this paper describes our difficult-path classification method and how to construct microthreads that will remove a significant number of hardware mispredictions, while keeping microthread overhead in check. In addition, our mechanism can be implemented entirely in hardware, unlike previous schemes that rely on profile-based, compile-time analysis. We present the details of this hardware implementation in Section 4.

This paper is organized as follows. Section 2 discusses prior research relevant to this paper. Section 3 describes our new approach of using difficult paths to guide microthreaded branch prediction. Section 4 describes the algorithms and hardware necessary to implement our difficult-path branch prediction scheme. Section 5 provides experimental results and analysis. Section 6 provides conclusions.

## 2. Related Work

Branch prediction using subordinate microthreads was first proposed by Chappell *et al.* as an application of the Simultaneous Subordinate Microthreading (SSMT) paradigm [2]. SSMT was proposed as a general method for leveraging spare execution capacity to benefit the primary thread. Subsequent authors have referred to subordinate microthreads as "helper threads." The original microthread branch prediction mechanism used a hand-generated microthread to exploit local correlation of difficult branches identified through profiling. Many concepts from this previous work carry over to this paper, such as the basic microthreading hardware and the means by which to communicate branch predictions to the primary thread. In this paper, we attack a larger set of branch mispredictions with an automated, run-time mechanism that constructs more accurate microthreads.

Zilles and Sohi [18] proposed using *speculative slices* to pre-compute branch conditions and prefetching addresses. Their method used profiling data to hand-construct backward slices of computation for instructions responsible for many branch mispredictions or cache misses. The processor executed these speculative slices as helper threads to generate branch predictions and prefetches. Backward slices could contain control-flow, and several speculative optimizations were suggested to improve slice performance. Hardware mechanisms were proposed to coordinate dynamic branch prediction instances with the front-end and to squash useless threads on incorrect control-flow paths. Our work differs in the following ways: we target mispredictions using difficult paths, our hardware-based mechanism does not rely on profiling and hand analysis to generate microthreads, we leverage run-time information to create more timely microthreads, and we present simpler mechanisms for aborting useless microthreads and communicating microthread predictions to the front-end.

Roth and Sohi [16] proposed a processor capable of using *data-driven threads* (DDT) to perform critical computations—chains of instructions that lead to a mispredicted branch or cache miss. The DDT threads were non-speculative and the values produced were capable of being integrated into the primary thread via register integration [15]. The construction of the threads was performed automatically at compile-time using profiling data to estimate when DDT construction would be useful. This scheme did not convey branch predictions to the front-end, but instead pre-computed the results of branches so that they could be integrated back into the primary thread at rename time, thus shrinking misprediction penalties but not removing them. Our mechanism targets mispredictions with difficult paths, does not rely on the compiler, and is not constrained by the non-speculative nature of the threads.

Farcy *et al.* proposed a mechanism to target highly mispredicted branches within *local loops* [6]. The computations leading up to applicable branches were duplicated at decode time and used to pre-compute the conditions of the branches several loop iterations ahead of the current iteration. In order to get ahead, their scheme used stride value prediction for live-input values. This paper also proposed a mechanism by which to communicate predictions to the appropriate later iterations of the local loop. Though clever, the applicability of their overall mechanism was limited only to local loop branches based on predictable live-input values.

Roth *et al.* also proposed hardware pre-computation mechanisms to predict virtual function call targets [13] and to prefetch linked data structures [14]. In these mechanisms, the machine detected specific instruction sequences leading to virtual function call targets or linked-list jump pointers. These mechanisms did not use separate threads to perform the computations, but instead mimicked them on a separate execution engine.

Several papers have recently investigated the use of pre-computation threads for prefetching. Collins, *et al.* proposed speculative pre-computation [4] and then dynamic speculative pre-computation [3]. The former used a profiling-based, compile-time mechanism to build simple address computations for load instructions that caused many cache misses. The follow-on paper proposed a hardware mechanism for dynamically capturing the computations. Luk proposed a mechanism for doing compiler-controlled prefetching using separate threads [12]. In this

mechanism, the machine would fork multiple speculative copies of the primary thread in order to prefetch irregular address patterns. Annavaram *et al.* proposed using a separate pre-computation engine to execute threads generated on-the-fly at fetch time [1]. This mechanism essentially prioritized computations for loads accounting for many cache misses.

*Assisted Execution* [5], proposed by Dubois and Song, is a multithreading paradigm that uses compiler-generated *nanothreads* in support of the primary thread. This previous work proposed a nanothread prefetching scheme and suggested other general ways which nanothreads could be used to improve single-threaded performance. This paper did not address branch prediction and suggested nanothreads as a means for the compiler to interact with a running program.

## 3. Difficult-Path Classification and Branch Prediction

In this paper, we refer to a *path* as a particular sequence of control-flow changes that lead to either a conditional or indirect *terminating branch*. We use the addresses of the $n$ taken branches prior to the terminating branch to specify the path. These $n$ addresses are combined in a shift-XOR hash to yield a path identifier, or $Path\_Id$.

Figure 1 shows all paths with $n \leq 2$ to terminating branch $A$ (paths leading away from $A$ are not shown). When $n = 1$, there are 2 paths: $BA$ and $EA$. For $n = 2$, there are 5 paths: $CBA$, $DBA$, $FEA$, $GEA$, and $HEA$. The $Path\_Id$ for each would be computed using a shift-XOR hash of the $n$ branch address. For example, the $Path\_Id$ of path $GEA$ would use the addresses of branches $G$ and $E$.

A *difficult path* has a terminating branch that is poorly predicted *when on that path*. More formally, given a threshold $T$, a path is difficult if its terminating branch has a misprediction rate greater than $T$ when on that path. Note that it is entirely possible (and desirable) that many other paths to the same terminating branch are not difficult.

An important concept related to control-flow paths is *scope*. We define the scope of a path to be the sequence of instructions that comprise the $n$ control-flow blocks of that path[2]. Figure 1 shows the scope of path $GEA$ in the shaded blocks. This set of instructions is guaranteed to execute each time path $GEA$ is taken to branch $A$. Note that the block containing branch $G$ is not part of the scope, since it could have multiple entry points that alter the sequence of instructions executed before the branch instruction $G$.

[2]This is similar to the idea of an instruction's "neighborhood" as defined in [6]. A branch's neighborhood was used as the set of instructions to be analyzed for detecting local loops.

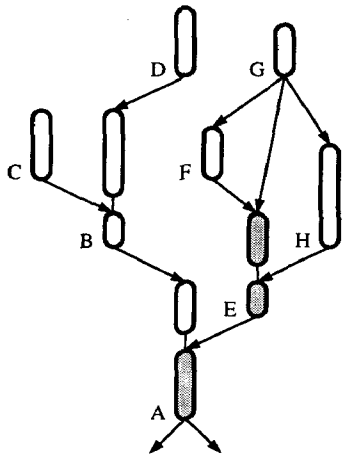


**Figure 1. Paths are identified by a terminating branch ($A$) and the last $n$ control-flow changes. Ovals indicate blocks adjacent along fall-through paths. Letters indicate taken branches. The scope of path $GEA$ is the set of instructions in the shaded blocks.**

### 3.1. Measuring Path Characteristics

We characterized the behavior of our chosen benchmarks in terms of paths and average scope as described in the previous section. The results are shown in Table 1 for several values of $n$.

As one would expect, the number of unique paths escalates quickly as $n$ increases. A larger value of $n$ results in the differentiation of several unique paths that would be considered a single path with a smaller value of $n$. Adjusting the value of $n$ adjusts the resolution at which paths are differentiated.

The average scope among unique paths also tends to increase with $n$. Paths get longer as more control-flow blocks are added. It is interesting to note that, with relatively small values of $n$, it is possible to produce paths with scopes in excess of 100 instructions.

The number of difficult paths does not change markedly as $T$ is varied between .05 and .15, especially with higher values of $n$. This is interesting, since it implies there is a fairly stable set of difficult paths that really are difficult.

### 3.2. Using Difficult Paths

Our goal is to improve machine performance via higher branch prediction accuracy. Previous research has demonstrated that more accurate branch predictions can be produced using subordinate microthreads. For these mechanisms to be successful, microthreads must target hardware mispredictions, compute accurate predictions, and complete

**Table 1. Unique paths, average scope size (in # of instructions), and number of difficult paths for different values of $n$ and $T$.**

| Bench | $n = 4$ | | | | | $n = 10$ | | | | | $n = 16$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | path | scope | $T$=.05 | $T$=.10 | $T$=.15 | path | scope | $T$=.05 | $T$=.10 | $T$=.15 | path | scope | $T$=.05 | $T$=.10 | $T$=.15 |
| comp | 1332 | 49.38 | 349 | 329 | 315 | 3320 | 123.77 | 723 | 682 | 646 | 8205 | 195.64 | 1307 | 1228 | 1145 |
| gcc | 131967 | 37.14 | 51259 | 41776 | 34942 | 428613 | 89.18 | 148513 | 129331 | 113982 | 886147 | 137.82 | 254463 | 229376 | 208212 |
| go | 113825 | 51.16 | 61526 | 54830 | 48127 | 681239 | 113.49 | 295722 | 273068 | 250213 | 1697537 | 171.80 | 589034 | 555140 | 519209 |
| ijpeg | 7679 | 62.98 | 1567 | 1263 | 1083 | 30624 | 153.64 | 7837 | 6873 | 6146 | 94023 | 228.17 | 21174 | 19401 | 17716 |
| li | 4095 | 36.16 | 576 | 491 | 457 | 8933 | 88.13 | 1401 | 1204 | 1093 | 16602 | 142.26 | 2615 | 2304 | 2084 |
| m88ksim | 5342 | 41.20 | 1266 | 1072 | 928 | 12397 | 99.60 | 2819 | 2486 | 2198 | 23460 | 164.51 | 4851 | 4445 | 4043 |
| perl | 11003 | 39.75 | 3109 | 3027 | 2926 | 26572 | 91.98 | 8116 | 7948 | 7717 | 47152 | 137.67 | 12311 | 12130 | 11929 |
| vortex | 36951 | 48.12 | 6231 | 4973 | 4415 | 76350 | 114.28 | 11929 | 9766 | 8672 | 119339 | 178.32 | 15672 | 13193 | 11779 |
| bzip2_2k | 23585 | 216.94 | 8861 | 6884 | 5685 | 836082 | 551.77 | 195652 | 180377 | 162349 | 4455846 | 541.59 | 935579 | 913986 | 882787 |
| crafty_2k | 59559 | 83.76 | 23225 | 18980 | 15806 | 361879 | 214.84 | 96830 | 86047 | 76964 | 942334 | 351.84 | 175022 | 159997 | 146174 |
| eon_2k | 15986 | 44.77 | 2584 | 2340 | 2147 | 32789 | 102.88 | 5021 | 4565 | 4182 | 48633 | 160.16 | 6540 | 5980 | 5493 |
| gap_2k | 28760 | 52.17 | 6883 | 5799 | 4966 | 84630 | 131.52 | 17855 | 15506 | 13455 | 165838 | 217.80 | 28742 | 25333 | 22332 |
| gcc_2k | 203334 | 55.63 | 75697 | 63754 | 54165 | 671250 | 132.41 | 185210 | 167533 | 151113 | 1191885 | 205.37 | 262718 | 244412 | 226077 |
| gzip_2k | 21942 | 100.94 | 9311 | 8091 | 7111 | 472396 | 267.46 | 118583 | 112095 | 105213 | 1973159 | 412.21 | 340683 | 332439 | 322094 |
| mcf_2k | 7707 | 46.05 | 2834 | 2387 | 2090 | 65498 | 118.08 | 17960 | 16357 | 15010 | 232125 | 165.48 | 45391 | 42793 | 40289 |
| parser_2k | 22174 | 49.65 | 8567 | 7851 | 7296 | 105758 | 119.59 | 29265 | 27014 | 25026 | 374747 | 181.99 | 74828 | 69928 | 65378 |
| perlbmk_2k | 12608 | 47.38 | 5145 | 5083 | 4996 | 22337 | 112.44 | 8108 | 8020 | 7920 | 28475 | 175.75 | 9207 | 9109 | 9011 |
| twolf_2k | 24280 | 62.46 | 7894 | 7097 | 6403 | 91321 | 162.95 | 23630 | 21395 | 19457 | 240853 | 251.63 | 48833 | 44970 | 41313 |
| vortex_2k | 57718 | 65.13 | 9285 | 8103 | 7384 | 130800 | 148.84 | 18813 | 16991 | 15820 | 208697 | 229.24 | 24619 | 22534 | 21086 |
| vpr_2k | 34589 | 111.11 | 10977 | 9586 | 8579 | 1330809 | 348.34 | 247932 | 240666 | 230405 | 4895234 | 550.59 | 616776 | 613795 | 608067 |
| Average | 41222 | 65.09 | 14857 | 12686 | 10991 | 273680 | 164.26 | 72096 | 66396 | 60879 | 882515 | 239.99 | 173518 | 166125 | 158311 |

in a timely manner. This section describes how, using difficult paths, we can construct microthreads to accomplish these goals.

**3.2.1. Targeting Mispredictions.** We wish to use microthreads only for branch instances likely to be mispredicted. As described in Section 1, any microthread spawned for a correctly predicted branch incurs useless overhead (note that such overhead is not *always* useless, if significant prefetching occurs). Any microthread spawned for a correctly predicted branch also risks introduction of a misprediction.

In practice, targeting mispredictions is somewhat complicated. Predictability must be considered at the time microthreads are constructed. Previous studies have targeted mispredictions simply by concentrating on static branches that exhibit poor predictability. We propose to use difficult paths instead.

A correlation exists between dynamic control-flow information and branch predictability [10]. Given this, it follows that a set of difficult paths can achieve greater "misprediction resolution" than a set of difficult branches. This also makes sense intuitively: difficult branches often have many easy paths, and easy branches often have a few difficult paths. By considering only the set of difficult paths, we eliminate a great number of branch instances.

Table 2 shows the misprediction and execution coverages for the SPECint95 and SPECint2000 benchmark suites for different values of $n$ and $T$. The same definition of "difficult" ($mispr\_rate > T$) applies to both branches and paths. The table shows that, generally, classifying by paths increases coverage of mispredictions, while lowering execution coverage.

**3.2.2. Accurate Microthreads.** The importance of accurate microthread predictions should be clear: if a microthread generates an incorrect prediction, it causes a misprediction recovery and lowers performance.

Previous research has shown that pre-computation threads can very accurately pre-compute branch conditions [6, 16, 18] (see Section 2). However, these mechanisms require hand-analysis or complex profiling to generate microthreads. Hand-analysis methods clearly have limited applicability. Previous profiling methods require analysis to consider and reconcile all possible paths to each difficult branch. The storage and complexity both scale with the control-flow depth considered.

We propose to construct a pre-computation microthread to predict the terminating branch of each difficult path. Because microthreads *pre-compute* the outcome of the branch, the predictions are very accurate. Because each microthread need predict the terminating branch for a single difficult path, the construction process is very simple.

To construct a microthread, we consider the *scope* of the difficult path (the set of instructions guaranteed to execute each time the path is encountered). By observing the dataflow within the scope, we can easily extract a subset of instructions that will pre-compute the branch condition and target address. This subset of instructions becomes the prediction microthread for the given difficult path. The construction process is simple enough to implement in hardware. Details are presented in Section 4.

**3.2.3. Timely Microthreads.** Microthread predictions must arrive in time to be useful. Ideally, every microthread would complete before the fetch of the corresponding branch. However, late predictions are still useful to ini-

**Table 2. Misprediction and execution coverages for difficult branches (Branch) and difficult paths ($n = \{4, 10, 16\}$). Each percentage represents the fraction of total mispredictions or dynamic branch executions covered by the set of difficult branches or paths.**

| | $T = .05$ | | | | | | | | $T = .10$ | | | | | | | | $T = .15$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Branch | | $n = 4$ | | $n = 10$ | | $n = 16$ | | Branch | | $n = 4$ | | $n = 10$ | | $n = 16$ | | Branch | | $n = 4$ | | $n = 10$ | | $n = 16$ | |
| Bench | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% | mis% | exe% |
| comp | 98.2 | 18.3 | 98.2 | 17.5 | 98.2 | 15.8 | 97.9 | 15.1 | 94.6 | 16.5 | 94.2 | 15.5 | 95.3 | 13.8 | 94.9 | 13.2 | 94.6 | 16.5 | 88.5 | 13.1 | 91.0 | 12.1 | 92.1 | 12.0 |
| gcc | 83.3 | 31.6 | 85.3 | 26.0 | 88.4 | 22.8 | 90.2 | 20.6 | 63.6 | 17.6 | 72.1 | 16.4 | 78.1 | 15.3 | 81.4 | 14.1 | 47.2 | 10.6 | 60.4 | 11.4 | 68.5 | 11.3 | 73.5 | 10.8 |
| go | 96.3 | 66.2 | 94.9 | 57.8 | 95.2 | 47.9 | 96.1 | 40.9 | 85.2 | 49.0 | 84.6 | 41.4 | 87.5 | 35.7 | 90.0 | 31.3 | 68.3 | 34.2 | 73.8 | 31.6 | 79.0 | 27.9 | 83.3 | 25.2 |
| ijpeg | 90.6 | 25.8 | 90.1 | 21.0 | 91.6 | 17.5 | 93.1 | 16.8 | 85.5 | 21.8 | 83.7 | 14.6 | 88.7 | 15.0 | 91.6 | 15.4 | 65.7 | 10.9 | 81.8 | 13.6 | 86.3 | 13.8 | 88.6 | 13.8 |
| li | 89.6 | 18.3 | 92.1 | 15.5 | 95.8 | 14.8 | 94.8 | 12.2 | 79.6 | 13.9 | 83.1 | 11.2 | 91.2 | 12.1 | 92.8 | 11.2 | 64.6 | 10.0 | 80.9 | 10.5 | 87.3 | 11.0 | 86.3 | 9.5 |
| m88ksim | 58.7 | 4.0 | 64.9 | 3.8 | 69.3 | 3.2 | 87.9 | 6.3 | 48.4 | 2.6 | 57.7 | 2.8 | 62.8 | 2.3 | 67.8 | 2.5 | 41.2 | 2.0 | 47.0 | 1.8 | 56.4 | 1.8 | 60.7 | 1.8 |
| perl | 68.4 | 6.1 | 90.7 | 7.7 | 95.7 | 5.2 | 97.0 | 4.3 | 58.2 | 4.2 | 71.6 | 3.7 | 91.0 | 4.1 | 94.1 | 3.7 | 38.5 | 1.8 | 63.5 | 2.8 | 86.8 | 3.6 | 90.7 | 3.3 |
| vortex | 75.8 | 4.0 | 81.2 | 3.0 | 87.6 | 2.9 | 90.8 | 2.7 | 61.2 | 2.7 | 72.7 | 2.2 | 78.5 | 2.1 | 83.6 | 2.1 | 34.4 | 1.1 | 59.3 | 1.4 | 68.3 | 1.5 | 73.1 | 1.5 |
| bzip2_2k | 96.8 | 38.5 | 96.0 | 33.2 | 96.0 | 29.2 | 97.0 | 23.5 | 91.7 | 32.5 | 91.7 | 28.1 | 90.5 | 23.1 | 93.4 | 19.1 | 81.4 | 25.5 | 84.6 | 23.3 | 85.5 | 19.6 | 90.2 | 17.0 |
| crafty_2k | 80.6 | 26.6 | 86.2 | 22.4 | 90.3 | 18.3 | 92.4 | 15.9 | 56.9 | 12.8 | 70.7 | 13.2 | 79.4 | 11.8 | 84.0 | 10.9 | 35.6 | 5.7 | 56.2 | 8.2 | 69.5 | 8.4 | 75.9 | 8.1 |
| eon_2k | 78.6 | 6.5 | 81.9 | 5.7 | 88.1 | 5.6 | 90.6 | 5.5 | 65.4 | 4.0 | 67.5 | 3.3 | 75.1 | 3.5 | 78.3 | 3.5 | 36.4 | 1.2 | 55.2 | 2.1 | 62.0 | 2.3 | 67.7 | 2.5 |
| gap_2k | 78.6 | 6.9 | 86.1 | 5.6 | 90.0 | 5.0 | 92.4 | 4.4 | 56.4 | 3.5 | 75.7 | 4.0 | 79.7 | 3.4 | 86.2 | 3.5 | 48.2 | 2.7 | 63.3 | 2.9 | 69.0 | 2.4 | 74.4 | 2.4 |
| gcc_2k | 84.0 | 31.7 | 88.9 | 26.4 | 91.1 | 20.7 | 93.5 | 19.0 | 66.7 | 18.7 | 76.7 | 16.7 | 83.4 | 14.8 | 86.5 | 13.5 | 49.1 | 11.0 | 65.6 | 11.9 | 75.1 | 11.1 | 79.8 | 10.5 |
| gzip_2k | 91.4 | 38.0 | 87.1 | 24.3 | 91.8 | 21.0 | 93.9 | 18.0 | 78.9 | 27.1 | 79.0 | 17.2 | 85.8 | 15.9 | 89.0 | 13.8 | 43.5 | 9.3 | 72.2 | 14.0 | 80.6 | 13.4 | 84.6 | 11.7 |
| mcf_2k | 73.5 | 21.6 | 84.6 | 21.2 | 83.9 | 15.3 | 85.1 | 13.1 | 47.7 | 9.8 | 62.2 | 10.6 | 66.1 | 7.3 | 73.6 | 7.2 | 40.6 | 7.9 | 34.5 | 3.5 | 59.0 | 5.5 | 68.0 | 5.7 |
| parser_2k | 85.7 | 21.0 | 94.1 | 22.2 | 94.0 | 17.6 | 95.4 | 16.6 | 78.9 | 16.9 | 84.2 | 15.8 | 88.9 | 14.2 | 90.2 | 13.1 | 67.7 | 12.7 | 69.4 | 10.0 | 79.0 | 10.6 | 83.8 | 10.8 |
| perlbmk_2k | 86.6 | 0.11 | 90.5 | 0.08 | 93.5 | 0.07 | 94.5 | 0.07 | 83.4 | 0.09 | 88.7 | 0.07 | 92.3 | 0.07 | 93.2 | 0.06 | 80.6 | 0.08 | 87.0 | 0.07 | 89.7 | 0.06 | 91.2 | 0.05 |
| twolf_2k | 91.7 | 22.9 | 95.8 | 21.2 | 96.5 | 18.0 | 97.0 | 16.6 | 87.8 | 20.0 | 91.1 | 17.5 | 92.9 | 15.1 | 93.8 | 14.0 | 79.4 | 16.3 | 84.5 | 14.3 | 88.8 | 13.1 | 90.6 | 12.5 |
| vortex_2k | 82.5 | 3.9 | 87.8 | 2.5 | 90.7 | 2.3 | 91.5 | 2.1 | 54.9 | 1.8 | 80.1 | 1.8 | 83.5 | 1.7 | 85.9 | 1.7 | 35.5 | 0.8 | 69.7 | 1.3 | 76.3 | 1.4 | 77.6 | 1.3 |
| vpr_2k | 90.9 | 28.9 | 96.5 | 30.7 | 98.4 | 23.5 | 99.2 | 14.2 | 87.5 | 24.4 | 91.6 | 25.0 | 96.3 | 21.1 | 98.4 | 13.3 | 85.0 | 22.8 | 85.4 | 20.8 | 92.5 | 18.6 | 96.8 | 12.2 |
| Average | 84.1 | 21.1 | 88.6 | 18.4 | 91.3 | 15.3 | 93.5 | 13.4 | 71.6 | 15.0 | 79.0 | 13.0 | 84.3 | 11.6 | 87.4 | 10.4 | 56.9 | 10.1 | 69.1 | 9.9 | 77.5 | 9.5 | 81.4 | 8.6 |

tiate early recoveries (we assume that microthread predictions will always be more accurate).

Timeliness requires two components: early spawns and quick microthread execution. Unfortunately, these two factors tend to work against each other—earlier spawns tend to require longer, and slower, microthread computations to pre-generate the branch outcome.

We can obtain earlier spawn points by increasing the scope of the difficult paths (by increasing $n$). This allows the microthread to be launched further "ahead" of the branch, while maintaining the important microthread characteristics described in the previous sections. There are downsides to doing this, such as increasing the number of unique paths and the number of extraneous spawns. These problems are adequately handled in our mechanism.

We propose to shorten microthread computations using a technique called *pruning*, which is applied at the time microthreads are constructed. Value and address predictability are known to exist in applications [11, 17]. We intend to prune sub-trees of computation by exploiting this predictability[3]. An example of pruning is shown in Figure 2.

Pruning requires two capabilities. First, the machine must identify predictable values and addresses at the time microthreads are being constructed. If this is done at compile-time, profiling information could be used. If this is done at run-time, this information must be tracked by the construction hardware.

[3]González and González proposed to use value speculation for the inputs to branch comparisons [8]. It was done at prediction time by a hardware mechanism. This is similar to pruning in that it shortcuts the branch predicate calculation.

Second, the machine must be able to dynamically generate value and address predictions for use in pruned microthread computations. To accomplish this, we add two new micro-instructions, *Vp_Inst* and *Ap_Inst*. Either of these instructions is used to replace each pruned sub-tree of computation. The machine executes these new instructions by querying special-purpose value and address predictors.

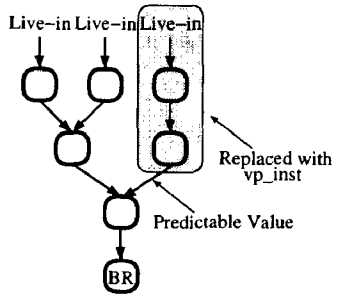


**Figure 2. An example of pruning.**

## 4. Building and Using Difficult-Path Microthreads

This section presents a hardware implementation of our mechanism. It includes structures to identify difficult paths, construct and optimize microthreads, and communicate microthread predictions to the primary thread. Compile-time implementations, which we have also investigated, are outside the scope of this paper.

## 4.1. Identifying Difficult Paths: The Path Cache

Our hardware mechanism for identifying difficult paths is straightforward. We assume that the front-end can trivially generate our $Path\_Id$ hash and associate the current value to each branch instruction as it is fetched. A back-end structure, called the *Path Cache*, maintains state to identify difficult paths.

The Path Cache is updated as follows. As each branch retires from the machine, its $Path\_Id$ is used to index the Path Cache and update the corresponding entry. Each Path Cache entry contains two counters: one for the number of occurrences of the path, and another for associated number of hardware mispredictions of the terminating branch.

We define a number of occurrences, called the *training interval*, over which to measure a path's difficulty. At the end of a training interval, the hardware misprediction rate represented by the counters is compared to the difficulty threshold $T$. A *Difficult* bit stored in each Path Cache entry is set to represent the current difficulty of the path, as determined during the last training interval. After the *Difficult* bit is updated, the occurrence and misprediction counters are reset to zero.

Allocation and replacement in the Path Cache is tuned to favor *difficult* paths. We allocate a new entry only if the current terminating branch was mispredicted by the hardware predictor. Because of this, roughly 45% of the possible allocations can be ignored for an 8K-entry Path Cache, leaving more space to track difficult paths. When a Path Cache entry must be replaced, we use a modified LRU scheme that favors entries without the *Difficult* bit set.

## 4.2. Building Microthreads for Difficult Paths

Our mechanism uses microthreads to predict the terminating branches of difficult paths. The Path Cache, described above, identifies difficult paths at run-time. Now we must build microthreads to predict them.

**4.2.1. Promotion and Demotion.** We refer to the decision to predict a difficult path with a microthread as *path promotion*. The opposite decision is called *path demotion*.

In the simplest case, promotion and demotion events should correspond to changes in a Path Cache entry's *Difficult* bit. When the *Difficult* bit transitions from 0 to 1, we promote the path. When the *Difficult* bit transitions from 1 to 0, we wish to demote the path. To keep track of which paths are promoted, we add a *Promoted* bit to each Path Cache entry.

The *promotion logic* is responsible for generating promotion requests. Each time a Path Cache entry is updated (ie. when a branch retires), the entry's *Difficult* and *Promoted* bits are examined. In the case that the *Difficult* bit is set, but the *Promoted* bit is not set, a request is sent to the *Microthread Builder* to begin construction. If the builder can satisfy the request, the *Promoted* bit is then set.

**4.2.2. The Basics of Building Microthreads.** We refer to the the hardware associated with generating microthreads as the *Microthread Builder*. Figure 3 shows a high-level diagram of the various components.

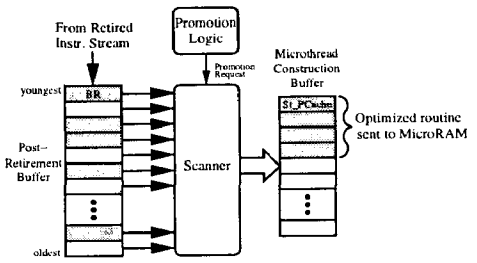


**Figure 3. The Microthread Builder**

The *Post-Retirement Buffer* (PRB) is used to store the last $i$ instructions to retire from the primary thread (we assume $i = 512$ in our implementation). Instructions enter the PRB after they retire and are pushed out as younger instructions are added. Dependency information, computed during instruction execution, is also stored in each PRB entry.

When a promotion request is received, the Microthread Builder extracts the data-flow tree needed to compute the branch outcome. The PRB is frozen and scanned from youngest to oldest (the branch will always be the youngest instruction, as it just retired). Instructions making up the data-flow tree are identified and extracted into the *Microthread Construction Buffer (MCB)*. The identification is not difficult, as the dependency information is already stored in the PRB. The basic extraction of the data-flow tree in this manner is similar to the mechanism in [3].

Termination of the data-flow tree occurs when any of the following conditions are satisfied: 1) the MCB fills up, 2) the next instruction being examined is outside the path's scope, or 3) a memory dependency is encountered (the store is not included). At this point, the MCB can be examined to turn the extracted data-flow tree into a microthread.

To create a functional microthread, we convert the terminating branch into a special $Store\_PCache$ microinstruction. When executed, the $Store\_PCache$ communicates the branch outcome generated by the microthread to the front-end of the machine. The communication takes place via the Prediction Cache (see Section 4.3.3).

The last step in in microthread construction is to select a *spawn point*. This is the point in the primary thread's execution that we wish the microthread to be injected into the machine—logically, a single program instruction. Choosing an effective spawn point is a difficult problem. In the current mechanism, we assume only that we wish to launch

the microthread as early as possible. As such, we choose the earliest instruction possible that is both within the path's scope and satisfies all of the microthread's live-in register and memory dependencies.

Our current design assumes there is only one Microthread Builder, and that it can construct only one thread at a time. Our experiments have shown that the microthread build latency, unless extreme, does not significantly influence performance.

**4.2.3. Basic Microthread Optimizations.** Move elimination and constant propagation are simple code optimizations we employ in the MCB to further improve the timeliness of our microthreads. We find that microthreads frequently span many control-flow levels in the program. As such, they tend contain many un-optimized sequences of code, many resulting from stack pointer manipulations or loop-carried variables. Hardware implementations of both of these optimizations have been previously proposed in fill-unit research [7]. Similar functionality could be installed in a hardware MCB.

**4.2.4. Memory Dependencies.** Memory dependencies also provide an opportunity for optimization. We terminate data-flow tree construction upon a memory dependency. The spawn point is chosen such that this memory dependency will be satisfied architecturally when the microthread is spawned. This assumes, pessimistically, that memory dependencies seen at construction time will always exist. The opposite case also occurs—if the memory dependency did not exist at construction time, it results in an optimistic speculation that there will never be a dependency in the future.

Our hardware mechanism naturally incorporates memory dependency speculation into the microthreads. The decision to speculate is simply based on the data-flow tree at construction time. We prevent over-speculation by rebuilding the microthread if a dependence violation occurs during microthread execution. When the microthread is rebuilt, the current mis-speculated dependency will be seen and incorporated into the new microthread.

A more advanced rebuilding approach might correct only speculations that cause *repeated* violations. We find that our simpler approach approximates this fairly well and requires almost no additional complexity.

**4.2.5. Pruning.** *Pruning*, introduced in Section 3.2.3, is an advanced optimization applied in the MCB that uses value and address predictability to eliminate sub-trees of computation within a microthread. When pruning is successful, the resulting microthread is smaller, has shorter dependency chains, and has fewer live-in dependencies.

To implement pruning, the machine must support the $Vp\_Inst$ and $Ap\_Inst$ micro-instructions. To provide this functionality, we add separate value and address predictors to the back-end of the processor. These predictors are trained on the primary thread's retirement stream just before the instructions enter the PRB. Since these predictors will not be integrated into the core, they can be kept apart from the critical sections of the chip.

The decision to prune is straightforward. We assume our value and address predictors have an integrated confidence mechanism. We access the current confidence and store it with each retired instruction in the PRB. When a microthread is constructed, instructions marked as confident represent pruning opportunities.

Pruning actually occurs in the MCB. Value-pruned instructions are removed from the MCB, along with the subtrees of data-flow leading up to them. In place of the removed data-flow, a $Vp\_Inst$ microinstruction is inserted to provide the output register value. Address-pruned instructions are treated similarly, except that the prunable load itself is not removed from the routine, and the $Ap\_Inst$ provides the address base register value.

When the microthread is spawned, the $Vp\_Inst$ and $Ap\_Inst$ microinstructions must contain all the information necessary to access the value and address predictors to receive a prediction. This process seems to be complicated by the fact that predictions must be made in advance of the progress of the primary thread (recall that the predictors are trained on retiring primary thread instructions). The distance between the spawn point and the instruction being predicted must be reconciled. This is actually simple to accomplish, since every microthread is tied to the scope of a particular path. At construction time, we need only compute the number of predictions that the $Vp\_Inst/Ap\_Inst$ is ahead. At execution time, this information is passed to the value or address predictor, which generates a prediction for the correct instance. Adapting the predictor design to support this operation is trivial, if we restrict our predictors to handle only constant and stride-based predictions. We assume this in our mechanism.

## 4.3. General SSMT Hardware

This section provides a brief overview of the general mechanism for spawning and simultaneously executing microthreads on an SSMT core. A more detailed description is not possible due to space limitations. We assume the general capabilities described in [2]. A high-level diagram of the core is shown in Figure 4.

**4.3.1. General Microthread Operation.** A microthread is invoked when its spawn point is fetched by the primary thread. If resources are available, the machine allocates a microcontext[4]for the newly spawned microthread. A spawn request is sent to the MicroRAM—the structure that stores

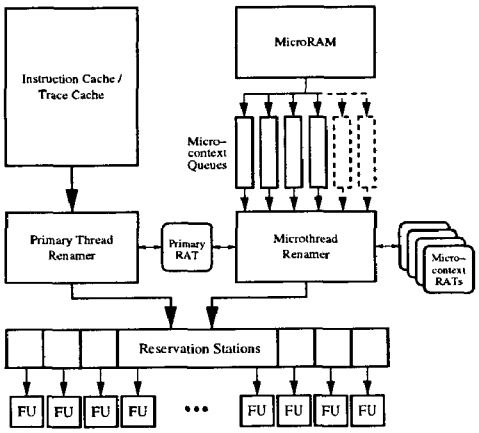


**Figure 4. Basic SSMT Processor Core**

SSMT routines. The MicroRAM delivers instructions from the specified routine into a microcontext queue. Each cycle, active microcontext queues are processed to build a packet of microthread instructions. These instructions are renamed and eventually issued to the reservations stations, where they execute out-of-order simultaneously with the primary thread. A microcontext is de-allocated when all instructions have drained from its issue queue.

**4.3.2. Abort Mechanism.** Our SSMT machine contains a mechanism to detect and abort useless microthreads. Microthreads are often spawned to predict branches on control-flow paths that the machine doesn't take. Our mechanism uses a concatenated path hash, called $Path\_History$, to detect when the machine deviates from the path predicted by an active microthread. When this occurs, these microthreads are aborted and the microcontext is reclaimed. We assume microthread instructions already in the out-of-order window cannot be aborted.

The abort mechanism is very important. Our machine is very wide and deep, which means spawns must be launched very early to stay ahead of the primary thread. Many useless spawns occur, but the abort mechanism is able to keep the overhead in check. On average, 67% of the attempted spawns are aborted before allocating a microcontext. 66% of successful spawns are aborted sometime before the microthread has completed.

**4.3.3. The Prediction Cache.** The Prediction Cache, originally proposed by Chappell *et al.* in [2], is the structure responsible for communicating branch predictions between microthreads and the primary thread. We have modified the Prediction Cache slightly from its original incarnation to support our path-based prediction scheme. Its operation within the front-end is summarized in Figure 5.

[4] A *microcontext*, proposed in [2], is the set of state associated with an active microthread.

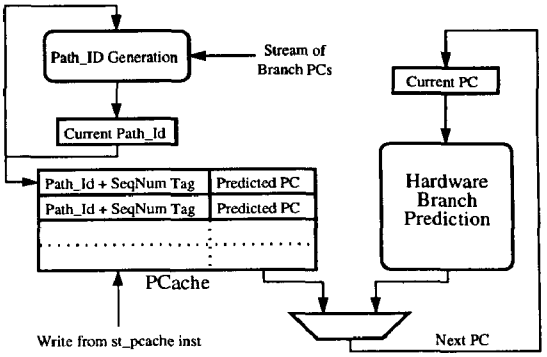


**Figure 5. Prediction Cache Operation**

A microthread writes the Prediction Cache using the $Path\_Id$ hash and the instruction sequence number, $Seq\_Num$[5], of the branch instance being predicted. The microthread computes the target branch $Seq\_Num$ by adding the predetermined instruction separation to the $Seq\_Num$ of the spawn. Because each ($Path\_Id$, $Seq\_Num$) pair specifies a particular instance of a branch on a particular path, our Prediction Cache naturally matches up microthread predictions written by $Store\_PCache$ instructions and the branches intended to consume them. Because both $Path\_Id$ and $Seq\_Num$ are used, aliasing is almost non-existent.

The ($Path\_Id$, $Seq\_Num$) pair is also used to match late microthread predictions with branch instances currently in-flight. If a late microthread prediction does not match the hardware prediction used for that branch, it is assumed that the microthread prediction is more accurate, and an early recovery is initiated. Because of the width and depth of our baseline machine, late predictions occur rather frequently.

The Prediction Cache does not need to maintain many concurrently active entries. Stale entries are easily de-allocated from the Prediction Cache by comparing the ($Path\_Id$, $Seq\_Num$) pair to the current position of the front-end. Because entries can be quickly de-allocated, the space can be more efficiently used. Our Prediction Cache can be made quite small (128 entries) with little impact on performance.

## 5. Performance Analysis

### 5.1. Machine Model

Our baseline configuration for these experiments modeled an aggressive wide-issue superscalar machine. The machine parameters are summarized in Table 3. All experiments were performed using the SPECint95 and

[5] An instruction sequence number, or $Seq\_Num$, is assigned to each instruction to represent its order within the dynamic instruction stream. Many machines already use sequence numbers for normal processing.

**Table 3. Baseline Machine Model**

| | |
|---|---|
| Fetch, Decode, Rename | 64KB, 4-way associative, instruction cache with 3 cycle latency capable of processing 3 accesses per cycle; 16-wide decoder with 1 cycle latency; 16-wide renamer with 4 cycle latency |
| Branch Predictors | 128K-entry gshare/PAs hybrid with 64K-entry hybrid selector; 4K-entry branch target buffer; 32-entry call/return stack; 64K-entry target cache (for indirect branches); all predictors capable of generating 3 predictions per cycle; total misprediction penalty is 20 cycles |
| Execution Core | 512-entry out-of-order window; physical register file has 4 cycle latency; 16 all-purpose functional units, fully-pipelined except for FP divide; full forwarding network; memory accesses scheduled using a perfect dependency predictor |
| Data Caches | 64KB, 2-way assoc L1 data cache with 3 cycle latency; 4 L1 cache read ports, 1 L2 write port, 8 L1 cache banks; 32-entry store/write-combining buffer; stores are sent directly to the L2 and invalidated in the L1; 64B-wide, full-speed L1/L2 bus; 1MB, 8-way associative L2 data cache with 6 cycle latency once access starts, 2 L2 read ports, 1 L2 write port, 8 L2 banks; caches use LRU replacement; all intermediate queues and traffic are modeled |
| Busses and Memory | memory controller on chip; 16 outstanding misses to memory; 32B-wide core to memory bus at 2:1 bus ratio; split address/data busses; 1 cycle bus arbitration; 100 cycle DRAM part access latency once access starts, 32 DRAM banks; all intermediate queues modeled |

SPECint2000 benchmark suites compiled for the Alpha EV6 ISA with `-fast` optimizations and profiling feedback enabled.

It is important to note that our experiments focused on improving an *aggressive* baseline. When using our approach, it is more difficult to improve performance when the primary thread already achieves high performance. Spawns must occur very early for microthreads to "stay ahead." This fact necessitates longer microthreads and causes many more useless spawns. This results in more overhead contention with the primary thread, despite the fact that our wide machine generally has more resources to spare.

Our machine also used an idealized front-end, also to avoid biasing our results. Microthreads take advantage of resources unused by the primary thread. A fetch bottleneck would unfairly under-utilize execution resources and leave more for the microthreads to consume. Our front-end can handle three branch predictions and three accesses to the instruction cache per cycle. In a sense, we are modeling a very efficient trace cache.

## 5.2. Potential of Difficult-Path Branch Prediction

Figure 6 shows the potential speed-up (in IPC) gained by perfectly predicting the terminating branches of difficult paths. Difficult paths were identified using $T = .10$ and $n = \{4, 10, 16\}$. We tracked difficult paths dynamically using an 8K-entry Path Cache and a training interval of 32. The MicroRAM size, which determines the number of concurrent promoted paths, was also set to 8K. We simulated many other configurations that we cannot report due to space limitations.

It is interesting that our potential speed-up was not closer to perfect branch prediction, since Table 2 suggests difficult paths have large misprediction coverage. However, Table 1 shows that benchmarks often have tens to hundreds of thousands of difficult paths. Our simple, realistic Path Cache simply could not track the sheer number of difficult paths well enough at run-time. Improving difficult path identification, both with the Path Cache and using the compiler, is an area of future work.

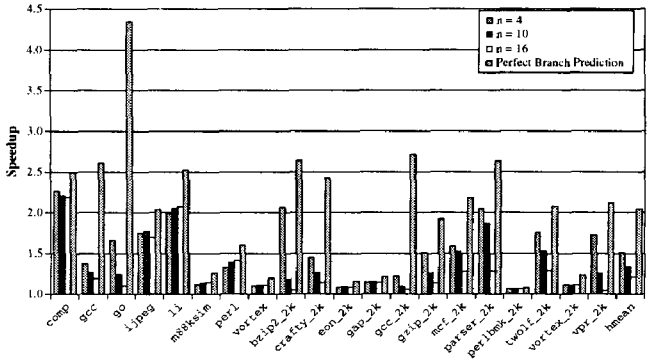


**Figure 6. Potential speed-up from perfect prediction** (8K-entry Path Cache, $T = .10$).

## 5.3. Realistic Performance

Figure 7 shows realistic machine speed-up when using our full mechanism. Speed-up is shown with and without the pruning optimization. Also shown is the speed-up when including microthread overhead, but not the microthread predictions. Parameters for difficult path identification were set as in the previous experiment. Microthread build latency was set to a fixed 100 cycles.

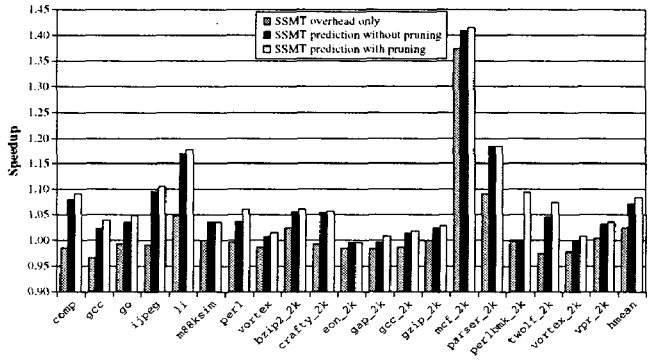


**Figure 7. Realistic speed-up** ($n = 10$, $T = .10$).

Our mechanism was successful at increasing performance in all benchmarks except eon_2k, which saw a slight loss. eon_2k and some other benchmarks are relatively well-behaved and do not have much tolerance for microthread overhead. Microthreads have a difficult time trying to "get ahead" of the front-end and compete more heavily for ex-

ecution resources. We are experimenting with feedback mechanisms to throttle microthread usage to address these problems.

Figure 7 also demonstrates the effectiveness of pruning. Pruning succeeded at increasing performance over our baseline microthread mechanism. We examine the reasons for this in the next section.

The remaining bar of Figure 7 shows speed-up due to microthread overhead alone. This measures the impact of overhead on the primary thread, without the positive effect of increased prediction accuracy. Pruning was disabled for this run. The majority of benchmarks saw a slight loss, which is to be expected. A couple benchmarks, notably mcf_2k, saw a significant gain. This can be attributed to prefetching effects from the microthread routines—a very pleasant side-effect.

## 5.4. Timeliness of Predictions

The pruning optimization increases performance by enabling smaller and faster microthread routines. This not only results in more timely microthread predictions, but also a smaller impact on the primary thread.

Figure 8 shows the average routine size and average longest dependency chain length of all routines generated with and without pruning. In general, pruning succeeded both at shortening microthread routines and reducing the critical dependency chains.

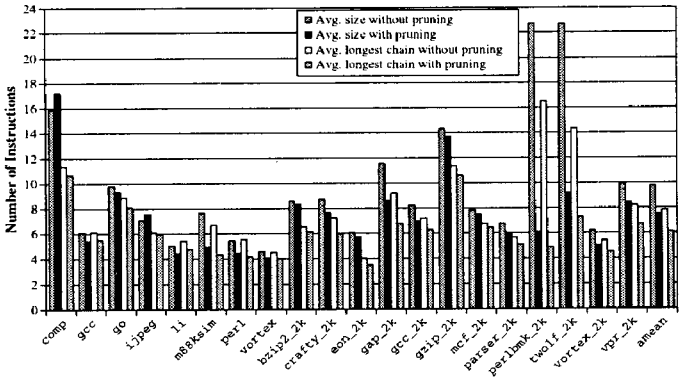


**Figure 8. Average routine size and average longest dependency chain length (in # insts).**

In a few interesting cases, such as compress, pruning increased the average routine length. This is because many live-in address base registers (typically global) were replaced by an *Ap_Inst* instruction, eliminating the live-in dependency but also lengthening the routine by one instruction. Even so, pruning was still successful at reducing the average longest dependency chain.

Microthread predictions can arrive before the branch is fetched (early), after the branch is fetched but before it is resolved (late), or after the branch is resolved (useless). Figure 9 shows the breakdown of prediction arrival times for our realistic configurations. Use of pruning resulted in an increased number of early predictions and useful (early + late) predictions. Use of pruning also slightly increased the overall number of predictions generated. This is because smaller microthreads free microcontexts more quickly, allowing more spawns to be processed.

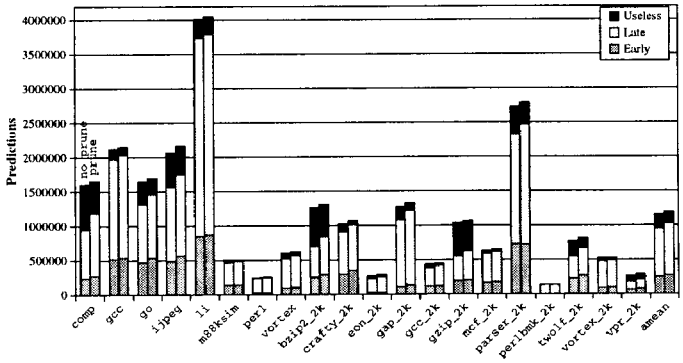


**Figure 9. Prediction timeliness broken down into early, late, and useless. The left and right bars represent no pruning and pruning, respectively. *Useless* does not include predictions for branches never reached.**

It is interesting to note from Figure 9 that, even with pruning, the majority of predictions still arrive after the branch is fetched. This is due, to some extent, to our idealistic fetch engine and rapid processing of the primary thread. However, it also indicates that there is still potential performance to be gained by further improving microthread timeliness.

## 6. Conclusions

Achieving accurate branch prediction remains a key challenge in future microprocessor designs. Previous research has proposed the use of subordinate microthreads to predict branches that are not handled effectively by known hardware schemes. Though microthread mechanisms have great potential for improving accuracy, past mechanisms have been limited by applicability and microthread overhead.

This paper proposes using difficult paths to improve prediction accuracy. We have shown how to build microthreads that better target hardware mispredictions, accurately predict branches, and compute predictions in time to remove some or all of the misprediction penalty. To demonstrate our approach, we have presented a hardware-only implementation of our scheme. We propose to identify difficult paths using the Path Cache, construct and optimize microthreads using the Microthread Builder, and communicate predictions to the primary thread using a modified Prediction Cache.

Because our mechanism can be implemented in hardware, we are not limited by compile-time assumptions. We do not depend on profiling data to construct our threads, and our mechanism can adapt during the run of a program. We have shown how our implementation can exploit runtime information to dynamically perform microthread optimizations. These optimizations include memory dependency speculation and pruning, a novel method of improving microthread latency based on value and address prediction.

Although this paper has shown our initial mechanism to be successful, there are many ways in which it could be improved. In particular, our future work includes ways to better track the often vast numbers of paths, further limit useless spawns, and further improve microthread timeliness.

## 7. Acknowledgments

Robert Chappell is a Michigan PhD student on an extended visit at The University of Texas at Austin. We gratefully acknowledge the Cockrell Foundation and Intel Corporation for his support. Francis Tseng's stipend is provided by an Intel fellowship. We also thank Intel for their continuing financial support and for providing most of the computing resources we enjoy at Texas. Finally, we are constantly mindful of the importance of our regular interaction with the other members of the HPS group and our associated research scientists, including Jared Stark and Stephen Melvin.

## References

[1] M. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 52 – 61, 2001.

[2] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 186 – 195, 1999.

[3] J. Collins, D. M. Tullsen, H. Wang, and J. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.

[4] J. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[5] M. Dubois and Y. Song. Assited execution. In *CENG Technical Report 98-25*, 1998.

[6] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–68, 1998.

[7] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[8] J. González and A. González. Control-flow speculation through value prediction for superscalar processors. In *Proceedings of the 1999 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1999.

[9] G. Hinton. Computer Architecture Seminar, The University of Texas at Austin, Nov. 2000.

[10] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 142–152, 1996.

[11] M. H. Lipasti, C. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 226–237, 1996.

[12] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40 – 50, 2001.

[13] A. Roth, A. Moshovos, and G. Sohi. Improving virtual function call target prediction via dependence-based pre-computation. In *Proceedings of the 1999 International Conference on Supercomputing*, 1999.

[14] A. Roth and G. Sohi. Effective jump pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

[15] A. Roth and G. Sohi. Register integration: A simple and efficient implementation of squash reuse. In *Proceedings of the 33th Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[16] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh IEEE International Symposium on High Performance Computer Architecture*, 2001.

[17] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–257, 1997.

[18] C. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.