PROCEEDING OF THE 1994 27TH ANNUAL
INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, NOV. 30–DEC. 2 (SAN JOSE, CA)

Using Branch Handling Hardware to Support Profile-Driven Optimization

Thomas M. Conte* Burzin A. Patel* J. Stan Cox†

*Department of Electrical and Computer Engineering
University of South Carolina
Columbia, South Carolina 29205

†Database and Compiler Technology
AT&T Global Information Solutions
Columbia, South Carolina 29170

Abstract

Profile-based optimizations can be used for instruction scheduling, loop scheduling, data preloading, function in-lining, and instruction cache performance enhancement. However, these techniques have not been embraced by software vendors because programs instrumented for profiling run 2–30 times slower, an awkward *compile-run-recompile* sequence is required, and a test input suite must be collected and validated for each program. This paper proposes using existing branch handling hardware to generate profile information in real time. Techniques are presented for both one-level and two-level branch hardware organizations. The approach produces high accuracy with small slowdown in execution (0.4%–4.6%). This allows a program to be profiled while it is used, eliminating the need for a test input suite. This practically removes the inconvenience of profiling. With contemporary processors driven increasingly by compiler support, hardware-based profiling is important for high-performance systems.

1 Introduction

Advanced compilers perform optimizations across block boundaries to increase instruction-level parallelism, enhance resource usage and improve cache performance. Many of these methods, such as trace scheduling [1], and superblock scheduling [2], either rely on or can benefit from information about dynamic program behavior. For example, traditional optimizations enhance performance by an additional 15% when combined with profile-driven superblock formation [2]. Other examples include data preloading [3], improved function in-lining [4], and improved instruction cache performance [5].

There are several drawbacks to profile-driven optimizations. Many of the techniques can result in code size explosion if they are performed too aggressively. Dynamic basic block execution frequencies can be used to reduce this phenomenon. More problematic is the task of profiling itself. Obtaining profile data through software methods can be complex and time consuming, requiring additional steps in the compilation process. The usual method employed is a *compile-run-recompile* sequence. First, the program is compiled with profiling probes placed within each basic block[1]. The program is then run using several different test inputs. The resulting profile data is used to drive a profile-based compilation of the original program.

Execution of the profiled version of the program is slow. With some methods, the profiled version runs 30 times slower than the optimized program. At best, a profiling program can be expected to run two times slower. In addition, test inputs need to be carefully chosen [6],[7].

Static estimation solves some of the problems related to gathering profile data [8]. However, these techniques are not as accurate as profiling [6],[7]. When used for superblock scheduling, static estimates achieve approximately 50% of the speedup that profiling can achieve [9].

Many commercial microprocessors, such as the Pentium series [10] and the PowerPC 604 [11], incorporate some form of branch handling hardware. This paper proposes using existing branch handling hardware, along with OS support, to obtain profile information. Using this, the slowdown for profiling is

0

[1]The profiling probes are extra instructions which log the execution of a basic block at run time.

imperceptible (e.g., 0.4%–4.6% increase). This allows an application to be deployed in the field and later retrieved for profile-based recompilation. Since it captures actual usage, it solves the problem of obtaining valid test inputs for profiling. It also allows profiling of real-time applications and system software. Using dynamic information improves the accuracy of static techniques. In general, the techniques presented in this paper solve many of the problems with profiling and expand the usefulness of profile-driven optimization.

The following section reviews several hardware branch prediction mechanisms, along with published mechanisms that out-perform those currently implemented. Methods for deriving profile information from hardware are discussed in the third section. Although these methods are less accurate than full-fledged profiling, they are significantly more accurate than static estimates. Metrics to measure this error are discussed in Section 3.4. The fourth section presents experimental results and discusses the tradeoffs between the various schemes. The paper closes with recommendations for hardware-supported profiling, many of which can be implemented today in existing systems.

2 Branch Prediction and Profiling

There are several contemporary dynamic branch prediction mechanisms that have been implemented in commercial processors. This section briefly reviews these schemes. A graph representation for profile information is also presented, along with two methods for grouping basic blocks into larger structures.

2.1 Contemporary branch handling mechanisms

There are two classes of branch prediction methods: *one-level* and *two-level* schemes. One-level schemes use the address of the branch instruction to index into a *branch target buffer* (BTB), which contains a small state machine for predicting the outcome of a branch. When the branch completes execution the actual outcome is used to update the state machine. Figure 1 depicts this process. The most common state machine for one-level schemes is the two-bit counter predictor, described in [12]. This predictor is implemented in several contemporary processors. The nominal size for the one-level branch prediction buffer is between 512 and 1024 entries. Our experiments show that the two-bit counter, when used with a 1024-entry BTB, achieves a branch prediction accuracy of 90% on-average across the SPEC92 benchmarks.

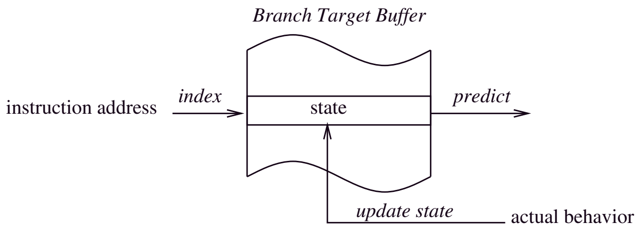

Figure 1: One-level branch prediction.

Two-level schemes use two separate buffers. The first buffer is indexed similar to the BTB and stores the branch history as a binary string. The second is indexed using this branch history and stores the state of a predictor. This is depicted in Figure 2. These schemes have been studied extensively by Yeh and Patt [13],[14] and we will use their nomenclature here. The first level buffer is termed the *history register table* (HRT). The HRT is b bits wide and stores a sequential, binary string of the branch's history, using 0 for not-taken and 1 for taken branches[2]. A prediction is made by indexing into the HRT, then using the history string to index into a second table, the *pattern table* (PT). The PT stores the state of a small state machine used to predict the branch. This decouples the branch prediction from the address of the branch instruction. The effect of this decoupling is dramatic. Yeh's algorithm can achieve 96% branch prediction accuracy for SPEC92 benchmarks [13],[14]. As of today, Yeh's algorithm has not been implemented in any commercially available microprocessor. However, the needs of wide-issue superscalars will likely drive future implementations of this branch predictor.

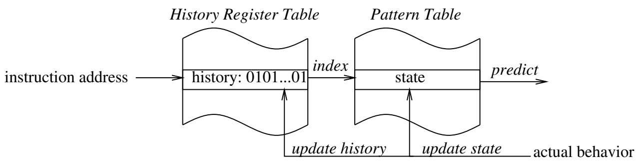

Figure 2: Two-level branch prediction.

2.2 Weighted control flow graphs

Profile-driven optimizations use a structure known as a *weighted control flow graph* (WCFG), which is a

[2] The "PAs" scheme with a 1024-entry HRT ($b = 12$) is used in this paper.

directed graph with basic-blocks as nodes. Arcs in a WCFG are due to one of two occurrences: either a code label or a branch instruction. An unweighted CFG for each function can be determined statically by the compiler.

A WCFG can be used to form larger groupings of blocks, which in turn can be used to enhance the scope of optimization and scheduling. Examples of these structures include Fisher's *traces* [1], and the IMPACT project *superblocks* [2]. Chang, *et al.* report a speedup of 15% when superblocks were used to extend the scope of traditional optimizations [2] Superblock formation and trace selection both use the same heuristics to form traces. Superblocks differ from traces in the method for providing fix-up code for off-trace/superblock execution and tail duplication [2],[9],[15]. Either method results in significant code size explosion. To limit this explosion, a threshold is placed on the execution frequency of a block. If a block's frequency is below this threshold, it is not considered for trace membership. (This is discussed in more detail in Section 3.4 below).

There are several methods of recording profile information. One method is to insert extra code at the beginning of each basic block that records the block id in a buffer. This buffer is then parsed into a WCFG, either periodically during execution, or after program completes execution. One example is the *Spike* profiler, which is built into the back end of GNU CC [16]. A disadvantage is its slowdown, which is approximately 30 times for *Spike*.

Another method used by AT&T Global Information Solutions in their commercial compilers is *arc-based profiling*. In this method, a transition block is added to the code to record the execution along an arc [17]. The target of the branch is changed to this new transition block, and an unconditional branch to the original destination is added to the end of the transition block. A table of all possible arcs is added to the object code by the compiler. An instruction to increment an arc's table entry is placed inside the transition block. When implemented, arc-based profiling results in a slowdown by a factor of two. Of the profiling approaches, arc-based profiling is the best suited to hardware adaptation.

2.3 The drawbacks of software profiling

Although the benefits of profile-driven optimization are large, there are many drawbacks to collecting profiles in software. The most severe is execution slowdown over unprofiled code. Slowdown is more than a minor inconvenience. Experience at AT&T Global Information Solutions has shown that slowdown is the major reason why profile-driven optimizations have not been adopted by the user community. Real-time applications such as kernels and embedded systems are excluded from the benefits of profile-driven optimizations. Long-running applications such as database systems are often excluded from profiling as well.

Another problem of profiling is the selection of inputs for the profiling task. Programs that are highly data-dependent, such as a sort routine (simple) or a database application (complex), have branches that are sensitive to user inputs. If the inputs are not selected carefully, the profile will not reflect actual usage. Validating profiling is difficult without a large scale study of user habits. In the absence of this, profiling is typically done using a large set of inputs, further increasing the time required for accurate profiling.

A third problem with profiling is the method for its use. A program must be compiled with profiling enabled, run using the test inputs, and then recompiled. For small programs, this is not difficult. For large systems, such as OS kernels or commercial database applications, this requires significant alteration of Makefile scripts [18]. A large amount of man-hours is invested in these scripts. For this reason, software vendors are hesitant to adopt any profile driven optimizations.

Ball and Larus have developed a set of heuristics to determine which arcs are more likely to be traversed in a CFG [8]. An extension to these that estimates node weights is presented in [19]. Such static heuristics can be used to solve many of the problems of profiling, but with less accurate results. For example, when static estimates are used to predict branch directions, the inaccuracies of the predictions are approximately twice that of profiled information [19].

3 Using Branch Prediction Hardware for Profiling

The goal of this paper is to demonstrate that the contents of hardware branch buffers can be used to add weights to a statically-built CFG. Most commercial processors allow the serial scan-out of state information for testing purposes. In addition to this, several processors implement kernel-mode instructions for reading branch hardware buffers directly. Hardware implementations typically include target address information along with prediction information. The combination of the target address (the destination of the arc), the buffer tag (the source of the arc), and the prediction information (the arc's weight), fully specify an arc in the WCFG.

The specific procedure for producing a WCFG is as follows: (1) A program is compiled with a special identifier token (magic number), indicating it contains a table of CFG arcs. (2) During execution, the kernel periodically reads the buffer and uses its contents to increment the arc counters. This period may be at every context switch, or more frequently. (3) On exit, the arc table is updated on disk. When branch hardware for profiling was implemented using a Pentium-based AT&T server system, results show an imperceptible difference in execution time between programs modified in this way and unmodified, traditionally-optimized programs. This slowdown is shown in Table 1 for five of the SPECint92 benchmarks. The maximum is for *espresso*, with a slowdown of just under 5%.

Table 1: Slowdown due to hardware profiling.

Benchmark	Unprofiled time (sec)	Hardware profiled time (sec)	Slow-down
compress	95.6	98.4	0.8%
eqntott	31.7	31.9	0.6%
espresso	45.4	47.6	4.6%
gcc	110.2	114.0	3.3%
xlisp	91.4	91.8	0.4%

3.1 Code adjustments to support arc-based profiling

There are two adjustments that need to be made to convert hardware branch information into arc weights. Indirect jumps can produce blocks with more than two outgoing arcs, reducing the one-to-one mapping between a buffer entry and an arc. The two primary sources of indirect jumps in C are due to call-through-pointers and *switch* statements. Call-through-pointers are not problematic, since trace selection is traditionally performed on a per-function basis. *Switch* statements can be converted into a chain of *if* statements. The performance lost from this conversion is later regained when the cases of the *switch* are sorted according to execution weight. The side-effect of this conversion is an increased branch target address predictability.

The second adjustment concerns code labels. Basic blocks formed due to code-labels are never allocated entries in the hardware buffer. A solution to this problem is to use the structure of the static graph to propagate profile information to these blocks. This can be done when the program is recompiled, and does not need to be done at execution time.

3.2 Two-level profiling

Slight modifications are required to adapt two-level schemes for the recording of arc weights. Since these buffers store a history of branch behavior, counting the number of 1's in a history register and dividing by the register width can be used to estimate the weight of an arc (we will refer to this as *dumping* the history register). After a history register is dumped, it must be updated in some fashion so that its contents are not over-counted at the next dumping point. Since dumping the buffer may occur at context switch points, there is no point in preserving the contents of the history registers. For these reasons, the history registers are initialized to 0 after being dumped[3].

Zeroing the history registers does not solve the problem of over-counting an arc's weight. Entries of '0' in the registers signify not-taken (fall-through) branches. A mechanism is needed that determines which '0's are due to actual execution and which are left over from the last buffer dump. Several techniques were experimented with, including marking each history with a *dirty bit*. In the dirty bit scheme, extra bits are added, one per history entry. These bits are cleared at each dump point. When a branch indexes into a history register, the bit is set. Since the HRT is a cache-like structure, it will contain a tag store. The "dirty" state can be marked by storing an invalid, impossible tag value for the history register entry. Thus, a *dirty bit* can be implemented with little hardware modification.

For frequently-encountered branches, the *dirty bit* scheme will produce accurate results. However, if a branch is encountered infrequently, the '0' entries from the buffer dump may still remain in a register, even though the *dirty bit* is set. This can increase the error for moderate-to-lightly visited basic blocks. The results in the following section support this claim.

Another scheme is to use a *marker bit* to record the boundary between the valid and invalid branch histories, as illustrated in Figure 3. After each dump, the registers are zeroed and the LSB of each register is set to '1' (i.e., '$00\cdots01$'). As the branch updates its history, this bit shifts to the left. An extra *full bit* is maintained in the MSB of the register. When *full bit* = 1, the entire contents of the register are treated as valid at a dump point. Otherwise, only the positions to the right of the leading '1' in the register are valid (i.e., if '$00\cdots01xxxx$', then only

[3] Our experiments show that this causes negligible change in the prediction accuracy. This is because context switches will normally result in a partial or near-total flush of the buffer.

the *xxxx* bits are valid). To complete this scheme, the *full bit* is zeroed at each dump point. Note that the *marker bit* is not wasted space. The entire history register holds a valid history once the *full bit* is set. Therefore, only one additional bit per history entry is required, regardless of the length of the HRT entries.

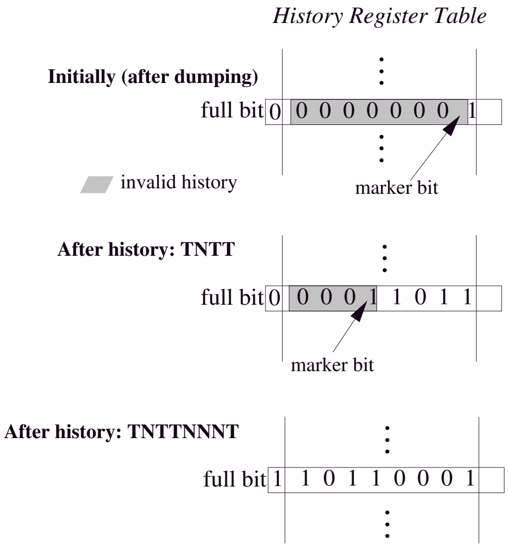

Figure 3: The marker bit modification to Yeh's algorithm for arc weight calculation.

3.3 One-level profiling

Modification of one-level schemes is less complicated, but also less accurate. Some indicator of the validity of a branch history is still required. For one-level schemes, this can be implemented using a *dirty bit*. As with the two-level schemes, this can be implemented without any modification if the BTB maintains a tag store.

The inaccuracy for one-level schemes is a result of using a two-bit counter to estimate the number of times a branch is taken or not-taken. As with the two-level scheme, several approaches were tried until one was found that achieved highly accurate performance. This scheme relies on the ability to keep a count of the total number of instructions executed since the last history dump. This is relatively easy since most modern processors have on-chip performance monitoring hardware to record such information[4]. Given that N instructions were executed, the compiler (and hence the dumping routine) can approximate the average number of instructions per basic block, $\bar{B}$. Then, if d entries in the BTB are dirty, each entry is assumed to be touched $\hat{w} = N/(\bar{B} \times d)$ times. This value is then used to translate the two-bit counter value into increments to the arc counters. This translation is presented in Table 2. The reason for the success of these approximations is discussed in Section 4 below.

[4] In the absence of monitoring hardware, an approximation can be obtained using the buffer sampling rate and the processor's CPI rating.

Table 2: Approximations used to convert BTB entries into arc weights.

Counter value	Value interpretation	Arc to increment	Increment value
00	strongly not-taken	fall-through	$2\hat{w}$
01	weakly not-taken	fall-through	$\hat{w}$
10	weakly taken	target	$\hat{w}$
11	strongly taken	target	$2\hat{w}$

(where $\hat{w} = N/(\bar{B} \times d)$)

3.4 Comparing profiles

Validation of hardware profiling is done by comparing traditionally-generated profiles (*actual profiles*) to hardware-generated profiles (*estimated profiles*). One method for this is to perform *trace selection* on both the actual and the estimated profiles and compare the results. An example of trace selection is illustrated in Figure 4. Graph (a) is annotated with the actual profile information, whereas graph (b) is the hardware-generated profile. Traces are formed using an arc trace selection threshold of 60% to group blocks [15]. Code explosion is avoided by not extending traces to blocks with low weights. This is implemented as a threshold, T. Values of T = 0.1%, 1%, 3% and 5% are considered below.

The metric for trace selection error is introduced using the example of Figure 4. In the actual graph (graph (a)), basic blocks **1**, **7**, **11** and **13** are grouped together to form a trace. Due to errors in the weights of outgoing arcs for block **7**, the blocks **1**, **7**, **8** and **9** are grouped to form a trace in the estimated graph. The error for block **7** is due to the difference in arc weights between the two graphs. The transition from block **7** to **8** will occur $0.15 \times 0.1 = 1.5\%$ of the total execution time. Similarly, the transition from **7** to **11** will occur $0.15 \times 0.9 = 13.5\%$ of the time. (Since the actual graph contains the real execution frequencies of the program, these frequencies are used.) Hence, the transition from **7** to **11** occupies a higher percentage of the total execution. The trace in graph (b) incorrectly assumes the transition of **7** to **8** is more likely. This assumption is wrong

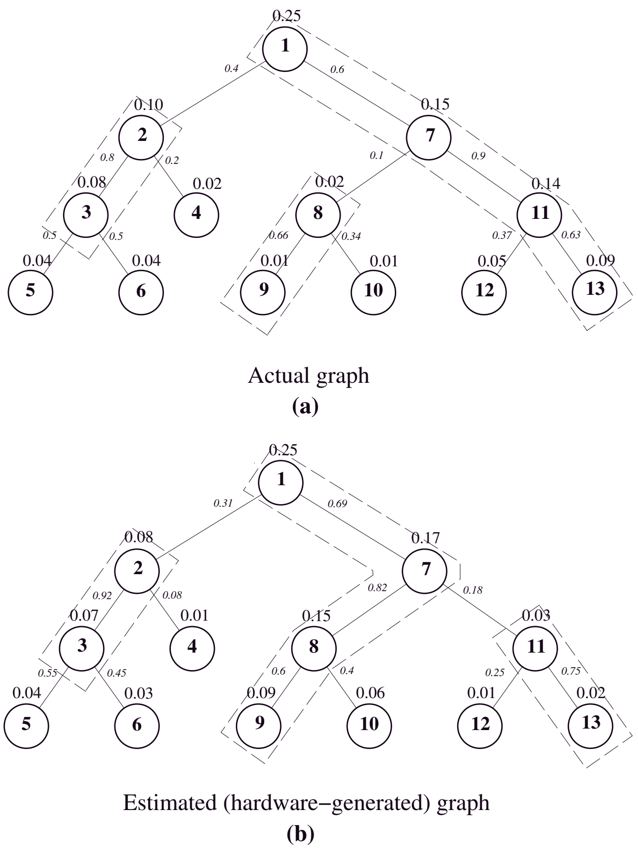

Figure 4: Trace selection example.

for $13.5\% - 1.5\% = 12\%$ of the execution. The figure of 12% is therefore the percentage of execution time that the incorrect trace membership will be exercised.

In general, the trace selection error is the total percentage of execution time that incorrect trace membership is exercised due to errors in the estimated profile. The trace selection error (TSE is formally specified by taking W_i to be the normalized weight (or, execution frequency) of block i. Let w_{ij} be the normalized weight of the arc from block i to block j. The trace selection error is calculated using the following procedure:

1. $TSE = 0$
2. **forall blocks** $i \in$ WCFG($actual$) **do:**
   (a) $j \leftarrow trace\_successor(\text{WCFG(actual)}, i)$
   (b) $k \leftarrow trace\_successor(\text{WCFG(estimate)}, i)$
   (c) **if** $j \neq k$ **then** $TSE \leftarrow TSE + W_i \times |w_{ij} - w_{ik}|$
3. **enddo**

Another method for comparison is the distribution of arc weight error versus block weights. This metric is useful since it shows where the trace selection error is occurring. The distribution is calculated by computing the maximum differences between the actual and the estimated arc weights for each category of block frequencies[5]. Let $\hat{w}_{ij}$ be the weight from i to j in the estimated (hardware-generated) profile, and w_{ij} be the weight for the actual profile. Define the maximum difference to be,

$$\Delta w_i = \max_{j \in \mathrm{succ}(i)} |w_{ij} - \hat{w}_{ij}|. \tag{1}$$

Then the (unnormalized) distribution function is,

$$f_{\mathrm{arcs}}(W) = \sum_{i \,\mathrm{s.t.}\, W_i = W} \Delta w_i, \tag{2}$$

or the sum of the maximum arc differences for each block with weight W. The distribution of arc weight error provides good insight into the performance of the techniques, as is shown in the next section.

4 Experimental results

The three schemes for hardware-based profiling were tested using benchmarks from the original SPEC92 benchmark suite as test workloads. Results are presented here for all the integer and an equal number of floating-point benchmarks (see Table 3). The

Table 3: SPEC92 benchmarks used for evaluation.

Class	Benchmark	Input
Integer	espresso	cps.in
	xlisp	li-input.lsp
	eqntott	int_pri_3.eqn
	compress	in
	sc	loada1
	gcc	tree.i
Floating-point	doduc	doducin
	nasa7	—
	mdljdp2	mdlj2.dat
	wave5	—
	tomcatv	—
	ora	params

benchmarks are compiled using the GNU C compiler with all optimizations enabled. The FORTRAN floating-point benchmarks are first converted from FORTRAN to C.

Several approximations are made in the previous section to extract arc weights from hardware. One

[5] The maximum difference is used in order to avoid overcounting a single error. For example, there is a 4% difference for two arcs with weights 40%/60% (actual) vs. 44%/56% (estimate), not an 8% difference.

Table 4: Dynamic basic block distribution.

Bench-mark	Basic Blocks E-25	E-50	E-90	E-99	E-100	Total static
espresso	15	49	225	842	2838	7582
xlisp	10	34	119	264	1058	3138
eqntott	1	2	6	34	502	1323
compress	2	5	17	21	135	432
sc	2	7	52	135	1529	4634
gcc	72	348	2610	6535	14382	34347
doduc	1	7	283	468	1596	3643
nasa7	2	2	2	2	210	1716
mdljdp2	2	5	15	35	821	848
wave5	2	14	72	177	1222	3896
tomcatv	3	6	12	14	372	1318
ora	3	6	13	24	396	1791

reason that these approximations are successful is the relatively high locality of branch instructions. This fact is illustrated in Table 4. These figures represent the distribution of unique basic blocks during execution. The "E-x" column presents the number of blocks that occupy x percent of the benchmark's execution. For example, of the 1323 branches in eqntott, only 502 are actually executed. Of these, only one branch accounts for 25% of the execution, and two branches for 50% of the execution. This table shows that most of the benchmarks exercise only a very small number of dynamic branches for the majority of their execution.

4.1 Performance of two-level profiling

Results for the two-level *dirty-bit* scheme are presented in Table 5. The columns labeled $T = y\%$ are for a code explosion cutoff threshold of y%. The trace selection error is remarkably low, even for a $T = 0.1\%$ cutoff. Several of the floating-point benchmarks achieve zero error. These benchmarks have a very low number of long-life dynamic branches, as shown in Table 4. The sources of the error are explained by the distribution of arc weight error, shown in Figure 6[6]. Notice that the majority of the difference in arc weights occurs for blocks that comprise less than 1% of the execution. This is true for all the benchmarks. This shows that the majority of the error for hardware profiling occurs for the lightly-executed blocks.

The code explosion cutoff threshold has a large effect on the error. Lower cutoff thresholds produce higher error for all schemes. This is because a hardware buffer only captures the most-frequently executed branches. Seldom-executed nodes and arcs will be poorly represented in the buffer. When the threshold is 3% to 5%, the error is zero in almost all of the cases. Two exceptions are integer benchmark *compress* and the floating-point benchmark *tomcatv*. For both these benchmarks, majority of the error is due to differences in two to three arc weights between the profiles.

[6] Only a subset of benchmarks are shown to simplify the graph, but their behavior is typical.

Table 5: Two level (dirty-bit) - trace selection error.

Benchmark	Trace selection error (percent) T = 0.1%	T = 1%	T = 3%	T = 5%
espresso	16.46	2.31	0	0
xlisp	12.23	5.69	0	0
eqntott	3.95	3.64	3.64	0
compress	16.16	15.31	6.12	6.12
sc	15.37	7.36	0	0
gcc	9.09	0	0	0
doduc	2.19	0	0	0
nasa7	0.21	0	0	0
mdljdp2	3.80	2.93	0	0
wave5	5.60	4.64	0	0
tomcatv	17.87	17.87	17.87	17.87
ora	0.02	0	0	0

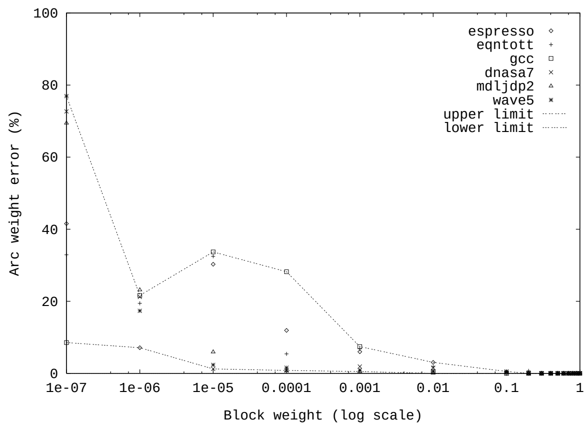

Figure 5: Two-level (*dirty-bit*): Distribution of arc weight error.

Closer examination of the error for all benchmarks suggests that the errors in trace selection have the effect of reducing the scope of the profile-based optimizations. Specifically, in *compress* a trace composed of blocks **33-36-37-38** in the actual profile was split into two traces between blocks **36** and **37** in the estimated profile. This occurred because of an error in the estimated arc frequency between blocks **36** and **37**. In this case the weight of this arc was less than the trace selection threshold, preventing the trace to grow beyond block **36**.

The primary reason that *tomcatv* experiences such high error is its lack of voluntary context switches (e.g., system calls). Because of this, the buffer is dumped only when the quantum expires. In the integer benchmarks, system calls are relatively frequent, causing a higher sampling rate. The effect of sampling the buffer more frequently than once a context switch is examined below in Section 4.3. Increasing the sampling rate reduces error without significant execution overhead[7].

Table 6: Two level (marker-bit) - trace selection error.

Benchmark	Trace selection error (percent)			
	T = 0.1%	T = 1%	T = 3%	T = 5%
espresso	12.67	2.31	0	0
xlisp	7.75	3.39	0	0
eqntott	3.95	3.64	3.64	0
compress	16.16	15.31	6.12	6.12
sc	8.46	6.40	0	0
gcc	2.39	0	0	0
doduc	1.21	0	0	0
nasa7	0.21	0	0	0
mdljdp2	3.80	2.93	0	0
wave5	2.32	2.32	0	0
tomcatv	17.87	17.87	17.87	17.87
ora	0	0	0	0

Implementation of the *marker-bit* scheme decreases trace selection error over *dirty-bit* for the majority of the benchmarks. For example, *espresso* drops from 16.46% to 12.67%. The marker improves the accuracy for lightly-executed basic blocks by increasing the accuracy of arc weights. This can be seen by comparing the *marker-bit* arc weight error distribution (Figure 6) to the distribution for the *dirty-bit* scheme (Figure 5). Observe that the error for lightly-weighted blocks has been significantly reduced, especially for the region between 10^{-5} and 0.001 (the pronounced error crest in Figure 5).

The *marker-bit* scheme does not help in all cases. This is especially true when the error-causing branches executed fairly frequently. This is the case for *compress*, *espresso* and *mdljdp2*.

[7]The execution overhead is approximately 0.55%.

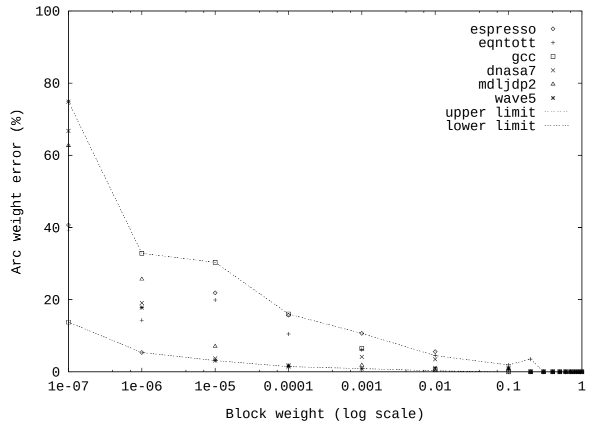

Figure 6: Two-level (*marker-bit*): Distribution of arc weight error.

4.2 Performance of one-level profiling

Profiling using one-level branch prediction hardware is often less accurate than two-level because the hardware contains a less-sophisticated measure of branch history. This results in a higher error (see Table ??). The benchmarks that perform poorly for two-level, perform poorly here as well. The arc weight error distribution (Figure 7) demonstrates why this is true. Even though the overall shape of the graph resembles the other two, there is a higher concentration of arc weight error between block weights 0.1 (10%) and 0.3 (30%). Because of this, some blocks that are relatively frequently accessed get incorrectly selected.

It is interesting to note that in some cases the one-level profiling is more accurate than the two-level schemes. This can be seen for benchmarks such as *sc*, *compress*, or *eqntott*. This is a consequence of estimating the block weight accurately. The weights are estimated using the techniques outlined in Table 2 of the previous section. This method predicts a branch's execution frequency based on how many instructions were executed since the last buffer dump. For the two-level schemes, the analog of this count is the width of the history register. This count saturates at 12, when the entire history register contains valid entries.

4.3 The effects of sampling rate on error

The results above show a relatively high error for benchmarks that sample the buffer only on quantum expiration. The worst case of this is the *tomcatv* benchmark. The effect of making the sampling rate

Table 7: One-level - trace selection error (percent).

Benchmark	Trace selection error T = 0.1%	T = 1%	T = 3%	T = 5%
espresso	18.49	5.51	0.02	0
xlisp	8.51	2.30	0	0
eqntott	3.66	3.64	3.64	0
compress	11.92	10.89	2.65	2.65
sc	3.46	1.14	0	0
gcc	5.99	0	0	0
doduc	2.86	0.14	0	0
nasa7	0.42	0.12	0.06	0
mdljdp2	2.80	2.20	0	0
wave5	7.01	6.06	0.07	0.07
tomcatv	25.28	25.19	24.34	24.34
ora	0.02	0	0	0

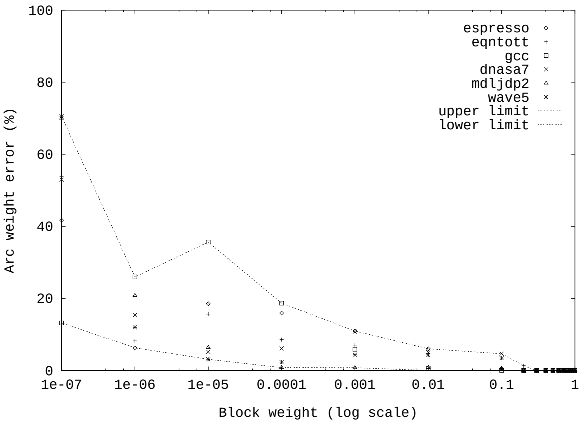

Figure 7: One-level: Distribution of arc weight error.

more frequent than the quantum are a reduction in error. This is presented in Table 8.

The error drops to zero for the two-level schemes when the sampling rate is increased to dump the buffer 16 times more frequently than the normal context switching rate. This is an important result, since it indicates a fast sampling rate can reduce even the highest error. Although the one-level approach improves with sampling rate, it does not go to zero. In general, if the scope of optimizations is limited due to trace selection error, the hardware buffer can be sampled more frequently.

Increasing the sampling rate does not appreciably affect execution time. For example, on a Pentium-based AT&T server the nominally hardware-profiled *tomcatv* takes 54.2 sec to execute, whereas interrupting *tomcatv* 16 times more frequently takes 54.5 sec

Table 8: The effects of sampling rate on error (*tomcatv*).

Sampling rate	Two-level (marker-bit) T = 0.1%	T = 1%	T = 3%	T = 5%
4×	17.87	17.87	17.87	17.87
8×	17.87	17.87	17.87	17.87
16×	0	0	0	0
	Two-level (dirty-bit)			
	T = 0.1%	T = 1%	T = 3%	T = 5%
4×	17.87	17.87	17.87	17.87
8×	17.87	17.87	17.87	17.87
16×	0	0	0	0
	One-level			
	T = 0.1%	T = 1%	T = 3%	T = 5%
4×	24.53	24.34	24.34	24.34
8×	19.85	19.85	17.90	17.90
16×	13.22	13.04	13.04	13.04

to execute[8]. The reason is that a buffer dump is not a full context switch. No change of context occurs. Sampling rate can be increased without significant performance impact, provided there is kernel support. The only performance degradation comes from flushing the buffer.

5 Concluding Remarks

This paper has presented a method for obtaining profile information without significant run-time slowdown (e.g., 0.4%–4.6%). This makes the *compile-use-recompile* approach presented here is much easier for software vendors than the traditional *compile-run-recompile* method of profiling. Using our techniques, software vendors can supply profiled versions of applications to alpha- and beta-testers, later collecting the profiles for final profiled optimizations. No sample suite of inputs is required. The longer the profiled version remains in the field, the higher the probability that the profiles match day-to-day use. Without the need for input sets, profiling can be used to optimize interactive, real-time, and system software packages.

All of the features required to use our techniques are already present in commercial processors. Most of these processors have branch target buffers, many that employ two-bit counter predictors. The trace selection error for these schemes is quite small. When the error does occur, it has the effect of limiting the scope of the optimization, which has few detrimental effects. It is important to note that the experimental

[8] Each of these experiments was run immediately after a reboot and the results are reproducible.

results are for a single run of each benchmark. The profiles are likely to converge after multiple runs, reducing the error still further.

Future superscalars will require branch prediction techniques more sophisticated than one-level BTB's, such as two-level approaches. Two schemes were presented to add hardware profiling to two-level mechanisms. Both schemes perform well for frequently-executed blocks. In addition, the *marker-bit* scheme performs well for moderately-executed blocks. The hardware overhead required for this scheme is minimal (specifically: one additional bit per HRT entry).

In general, the techniques presented here significantly reduce the inconvenience of profiling. With contemporary microarchitectures driven increasingly by compiler support, hardware-based profiling is important for continued improvements in processor performance.

Acknowledgements

This research has been supported by AT&T Global Information Solutions, a subsidiary of the AT&T Corporation.

References

[1] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

[2] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software–Practice and Experience*, vol. 21, pp. 1301–1321, Dec. 1991.

[3] W. Y. Chen, *Data preload for superscalar and VLIW processors.* PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, IL, 1993.

[4] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling C programs," in *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, (Portland, OR), June 1989.

[5] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Ann. International Symposium Computer Architecture*, (Jerusalem, Israel), pp. 242–251, May 1989.

[6] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proc. 5th Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, (Boston, MA), pp. 85–95, Oct. 1992.

[7] D. Wall, "Predicting program behavior using real or estimated profiles," in *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, (Toronto, Ontario, Canada), pp. 59–70, June 1991.

[8] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, (Albuquerque, NM), pp. 300–313, June 1993.

[9] R. E. Hank, S. A. Mahlke, J. C. Gyllenhaal, R. Bringmann, and W. W. Hwu, "Superblock formation using static program analysis," in *Proc. 26th Ann. Int'l. Symp. on Microarchitecture*, (Austin, TX), pp. 247–255, Dec. 1993.

[10] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.

[11] S. P. Song and M. Denman, "The PowerPC 604 RISC microprocessor," tech. rep., Somerset Design Center, Austin, TX, Apr. 1994.

[12] J. E. Smith, "A study of branch predition strategies," in *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, pp. 135–148, June 1981.

[13] T. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. 24th Ann. International Symposium on Microarchitecture*, (Albuquerque, NM), pp. 51–61, Nov. 1991.

[14] T. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proc. 20th Ann. International Symposium Computer Architecture*, (Ann Arbor, Michigan), pp. 257–266, May 1993.

[15] W. W. Hwu and P. P. Chang, "Trace selection for compiling large C application programs to microcode," in *Proc. 21st Ann. Workshop on Microprogramming and Microarchitectures*, (San Diego, CA.), Nov. 1988.

[16] M. L. Golden, "Issues in trace collection through program instrumentation," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 1991.

[17] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," Tech. Rep. 1031, Computer Sciences Dept., University of Wisconsin-Madison, 1991.

[18] S. I. Feldman, "Make - A program for maintaining computer programs," *Software–Practice and Experience*, vol. 9, pp. 255–265, Apr. 1979.

[19] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, "Accurate static estimators for program optimization," in *Proc. 6th Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, (Orlando, FL), pp. 85–95, June 1994.