Abstract

MENEZES, KISHORE NOEL PAUL

Hardware-Based Profiling for Program Optimization.

(Under the direction of Thomas M. Conte)

The process of characterizing the behavior of a program is known as *profiling*. The information about a program, obtained through this process, is called *profile information*. Traditionally, profiling is achieved through a long, tedious *instrument–run–recompile* sequence. During instrumentation, the compiler inserts additional instructions into the original program to collect accurate execution frequencies of basic blocks or the arcs that connect these basic blocks. Not only do these traditional techniques suffer from the need for multiple execution and compilation passes, they also suffer from slowdown of the program being profiled due to the added instructions. This thesis investigates hardware-based methods for obtaining profile information. Control flow profile information is collected using branch prediction hardware which is common in contemporary processors. The contents of the branch prediction hardware maintain information about the branches in code. This thesis suggests the use of this information for obtaining profiles of the branches in a program. The effectiveness of this kind of information when used in compiler optimizations is also investigated by applying such information to a prominent optimization, namely superblock scheduling. Methods of profiling for both one-level and two-level branch predictors, and the quality of information they yield are presented in the thesis. The use of custom hard-

ware for the purpose of profiling is also investigated. Statistical analysis is used to validate the results from hardware-based profiling. Hardware structures for memory dependence profiling are suggested and preliminary results presented.

Hardware-Based Profiling for Program Optimization

by

Kishore N. P. Menezes

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh

1997

Approved By:

Dr. Thomas M. Conte Chair of Advisory Committee	Dr. Paul D. Franzon
Dr. Alexandre E. Eichenberger	Dr. S. Purushothaman Iyer

Biography

Kishore Noel Paul Menezes was born June 26, 1969 in Bappalige in the state of Karnataka in India. Since then his parents made their home in Bombay. Kishore graduated high school from Don Bosco Apostolic School in Lonavla near Bombay in 1984. He attended junior college at G. N. Khalsa College in Bombay and graduated in 1986.

Kishore received the Bachelor of Engineering (B.E.) degree from the University of Bombay in 1990, after which he began work at Neutron Electronics, Bombay. He attended graduate school at the University of South Carolina in August 1992 and obtained the M.S. degree in Electrical and Computer Engineering in 1995. He then went on to enroll in the doctoral program in the Department of Electrical and Computer Engineering at North Carolina State University. On completion of his Ph.D., Kishore will join Intel Corporation (Santa Clara, CA).

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The process of characterizing the behavior of a program is known as *profiling* [1]. The information about a program, obtained through this process, is called *profile information* or simply a *profile*. Profile information has many applications. The more important of these pertain to optimizing the execution of the program to which the profile belongs. Some of the earliest work on profiling concentrated on obtaining information about the execution times of the functions within a given program [1]. The same could also be applied to algorithms. The statistics gathered were then used to optimize the functions with the highest execution times or to make improvements to badly implemented algorithms.

The scope and applications of profiling have changed over the years. Profile information is steadily gaining importance in the optimization of program execution, especially in the areas of hand–tuning of programs [1], trace scheduling [2], superblock

scheduling [3], data preloading [4], branch prediction [5], and improved instruction cache performance [3]. Issues in the exploitation of instruction–level parallelism inherent in programs, coupled with rapid developments in compiler research have generated interest in the role of profile information in smart compilation [6], [3], [7], [2], [8].

Program execution efficiency on modern day processors is severely limited by the control flow of a program. Advanced compilers perform optimizations across basic block boundaries to increase instruction-level parallelism, enhance resource usage and improve cache performance. Many of these methods, such as trace scheduling [2], and superblock scheduling [3], either rely on or can benefit from information about dynamic program behavior. An estimate of the more frequently executed portions of code in a program can be obtained from its profile. Compilers can then use this information to generate better code through educated optimization. For example, traditional optimizations enhance performance by an additional 15% when combined with profile-driven superblock formation [3]. Other examples include data preloading [4], improved function in-lining [9], and improved instruction cache performance [10]. Despite the advantages that have been shown to accrue from profiling, the cost associated with it has deterred many. Cost-effective solutions have been suggested [11] [12]. Considering the effectiveness of profiling in smarter compilation and its wide range of applications, a study of this topic is important. Subsection 1.1 describes some of the types of profiles possible. Subsection 1.2 describes some compiler optimizations using profile information suggested in the literature. A study of these optimizations

helps to better understand the information required in a profile. The need for the topic of this work, hardware based profiling, is explained in section 1.3. The chapter concludes with the contributions of the research conducted and documented in this thesis.

1.1 Types of profiles

Profiles, characterizations of a program, have been classified in a number of ways. One distinction drawn is between *static* and *dynamic* [13] profiles. Static profiles [6] [7] are obtained by examining the program itself. Educated guesses can be made, about the behavior of a program, by analyzing portions of it. (One example is the branch at the end of a loop. Such a branch can be expected to be *taken* most of the time.) Since static profiles are based on intuitive guesses rather than actual knowledge of the behavior of the program, the accuracy of such schemes is low. Dynamic profiles [14] [1] [12] are collected during the run-time of a program. They measure the execution behavior of a program and therefore provide more accurate profiles. However, the overhead associated with dynamic profiles is high. They require a compile-run-recompile sequence. They also require instruction probes or a call to a record-keeping function to collect the profile information. This increases the time required to collect a profile.

Profiles have also been classified based on the method of profile collection employed. Node-based profiles are collected by adding counting code to the basic blocks

(nodes) in a program. One example is the *Spike* profiler, which is built into the back end of GNU CC [15]. Other examples include *pixie* [16], and QPT [17]. When counting code is added along the arcs of a program flow graph the profile obtained is an arc-based profile. In this method, a transition block is added to the code to record the execution along an arc [18]. 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. The contents of a profile may also be used for classification. A profile which depicts the control behavior of a program may be termed a control profile. One that depicts the data behavior of the program may be termed a data profile.

1.2 Applications of profiling

The effectiveness of a profile is determined by the manner in which it is employed in the optimization of programs. The examination of applications that require profile information is important. Different optimizations use different types of profile information and in different ways. For example, a register allocation optimization may need information about global variables, while superblock scheduling uses information about basic blocks. The following subsections study some of the applications of profiling information. The requirements from profiling for these optimizations and how they can be obtained is also examined.

Scheduling

Instruction scheduling is one of the most important phases of the compiler [19], [20]. Performance can be greatly enhanced by an efficient instruction scheduler. For many years, research on instruction scheduling concentrated on the instructions within a basic block. The limited number of instructions within a basic block has prompted researchers to find parallelism across basic blocks. Conditional branches between basic blocks have been an hindrance. Trace scheduling [2] overcomes this problem by forming large traces from basic blocks that tend to execute together. Some knowledge about the execution frequencies of the basic blocks in the program is required. Fisher suggested that the execution frequencies of basic blocks be determined through profiling [2]. A long trace could be formed by picking the block with the highest frequency and then growing a trace above and below it. Efficient scheduling through code motion can then be performed within this trace. Code motion across branches within a trace requires patch-up code on the off-trace transitions. Profile information is important in such a method to help make decisions as to which basic block to include in the trace. In the absence of profile information a block with low actual execution frequency might be included in the trace thus causing an off-trace transition and penalizing the execution time of the program.

Ellis [21] suggested *arc-based* profiling as an alternative method of collecting profile information. *Arc-based* profiling computes the weight of arcs rather than that of the nodes in a control flow graph. Chang and Hwu [22] show that trace selection based

on the arc weights in a weighted control flow graph is indeed better than that based on node weights.

Hwu *et. al.* [23] describe a trace formation method which eliminates the book-keeping costs in trace-scheduling. The trace thus formed is called a *superblock*. Profile information is used to form traces in a manner similar to trace-scheduling. Superblock formation differs from the former method in that tail-duplication is performed on the traces that are formed. Tail-duplication involves duplicating the code that forms the tail of a trace so as to yield a trace with a single entrance but multiple exits. Tail-duplication eliminates the patch-up code required in off-trace blocks that branch into the trace. Accurate profile information is required for superblock formation since indiscriminate tail-duplication can lead to code explosion. Hank *et. al.* describe superblock formation using static estimation techniques. The implementation of traditional optimizations and the performance it yields using superblocks is the topic of [3].

In the search for larger scopes of scheduling and straight line code, predicated execution [24] [25] [26] has been suggested as a solution. Predicated execution forms large scopes of straight line code called a *hyperblock* in [25]. Each instruction in the hyperblock is predicated on the result of the branch or set of branches that precede it in the control flow graph. During execution the instruction is executed if the predicate it depends on is asserted, else it is squashed. A penalty is incurred for instructions that need to be squashed. Therefore, it is important that hyperblocks are not formed

indiscriminately. Profile information can be of great assistance in the decisions that need to be made during hyperblock formation.

I-cache layout

The primary aim of laying out code in the I-cache is to reduce the number of misses and the memory traffic. Using profile information to arrange code in the I-cache can greatly reduce the number of *conflict* misses and thus increase performance. This is especially true of large applications. Two functions, one called from the other may happen to share the same space in the I-cache. This can lead to an inordinate number of misses every time the function is called and returned from. Conflict misses can also be caused by memory references for code within a given function. This phenomenon occurs for large functions that may not fit completely in the I-cache.

Use of profile information to efficiently layout code in the I-cache is treated in [10]. The algorithm suggested in the paper uses a weighted call graph as well as a weighted control graph and consists of five steps.

1. Profiling: A weighted call graph is constructed by computing the actual execution frequencies of the nodes and arcs in the call graph. For each function in the call graph, a weighted control graph is constructed by computing execution frequencies for the basic blocks and the transitions between them.

2. Function inlining: The weighted call graph is used to determine functions with the highest execution frequency. These functions are then systematically in-

lined. Inline expansion increases the spatial locality as well as reduces cache mapping conflicts among functions.

3. Trace selection: The weighted control graph for each function is used to group the basic blocks that tend to execute together. This trace formation increases sequential and spatial locality.

4. Function layout: The traces are sorted in decreasing order of execution counts. The more-frequently executed traces are moved to the head of the function such that they are effectively in a single page or in adjacent pages.

5. Global layout: To further reduce the probability of two functions mapping to the same location in memory, the functions themselves are sorted by execution frequencies. The more frequently executed functions are then laid out in neighboring pages.

Load latency prediction

Processor performance is degraded by long cache-miss latencies. This is due to many different reasons. Code is normally scheduled for a processor by using one of two latencies for a load or store instruction: *best-case* or *worst-case*. In a hierarchical memory subsystem with many different levels the *best-case* latency is the first-level cache hit latency. The *worst-case* latency is the latency for a load/store instruction that misses in every level of the cache hierarchy. The variable latency of a load/store instruction in modern processors which use multiple levels in the cache hierarchy de-

tract from the smart schedule generated by the compiler. When code is scheduled using the *best-case* latency, a load/store that misses can delay the instructions dependent on it. Code scheduled with the *worst-case* latency requires instructions that can cover this latency. Such instructions are called load delay slots. However, using load delay slots for every load/store instruction in the program may result in an inefficient schedule.

Many different solutions have been suggested to handle the variable latency of a load/store instruction. Abraham *et. al.* [27] suggest a technique whereby the load/store instruction is tagged with specifiers indicating the expected load latency and the level in the cache hierarchy that the compiler expects the data to be in. Such information has to be gleaned from the program by the compiler. This would require heuristics or profile information. Cache-simulation based profiling is used to record the miss rate for each unique reference to the cache and the level of the cache it misses in. These statistics are then used to label a load/store with the information required by the processor to prefetch the required data into the appropriate level in the cache hierarchy.

1.3 Need for hardware-based profiling

Traditionally, profiling and the use of profiles is achieved through a long, tedious *instrument–run–recompile* sequence. During instrumentation, the compiler inserts additional instructions into the original program to collect accurate execution fre-

quencies of basic blocks or the arcs that connect these basic blocks. Next, this instrumented code is executed with a variety of batch inputs. After these multiple executions, the program statistics are calculated based on the profile information that has been collected. Finally, the original program is recompiled using profile-driven optimizations. Not only do these traditional techniques suffer from the need for multiple execution and compilation passes, they also suffer from slowdown of the program being profiled due to the added instructions. The MIPS basic block profiling tool, *pixie* [28], inserts about five instructions in every basic block [29]. Ball and Larus measured the slowdown of *pixie* required for arc–based profiling as between 1.11 to 5.24 times [29]. To offset this slowdown, modifications involving the reduction in the number of basic blocks or the arcs that need to be probed have been suggested in the literature [29], [6], [30], [31]. Ball and Larus investigated one such method which reduces the slowdown to a maximum of 2.05 times for the SPEC92 benchmarks. Unfortunately, this overhead is still too large for integrated software vendors to readily absorb. Commercial software vendors in general can tolerate a negligible amount of additional execution overhead (i.e., approximately $\leq 5\%$) [11]. Code instrumentation techniques, therefore, may not be appropriate for real–time, interactive, and transaction processing applications.

Static estimation solves some of the problems related to gathering profile data [6]. However, these techniques are not as accurate as dynamic profiling [14],[13]. When used for superblock scheduling, static estimates achieve approximately half of the

speedup that dynamic profiling can achieve [7]. Moreover, static estimation can only predict the direction of branches, not the relative execution frequencies of the paths. For example, a branch may be predicted *likely-taken*, but the number of times control would reach this branch is unknown. Optimizations based on such information are susceptible to code bloat. Actual execution frequencies can aid in making better decisions during trace generation for optimization.

If profiling is to gain commercial acceptance, it must be smoothly integrated into the software development cycle. Unfortunately, this requires the reduction or elimination of the need for a sample input suite, as well as more efficient profiling methods. *Hardware-based profiling* was introduced in Conte, *et al.* [12] as a way to address these problems. This technique uses existing branch prediction hardware to collect profile information at kernel entrances. It has a slowdown of 1.02 on average, and 1.05 as a worst case [12]. The use of hardware-based profiling allows software vendors to supply instrumented versions of applications to alpha and beta testers. The profiled information based on actual day to day usage can be later retrieved, and final program optimization can be performed. The advantage to hardware-based profiling is twofold: It eliminates the need for sample input suites (since actual usage can be captured), and the optimizations are based on actual program usage. Profiling in this manner is commercially appealing because vendors' alpha and beta testing processes are often very well defined, and the hardware-based style of profiling leverages their existing investment to produce better optimized code [11]. Since actual usage pat-

terns are profiled, the method also solves the problem of obtaining valid test inputs for profiling. This replaces the *compile-run-recompile* sequence with a less awkward *compile-use-recompile* sequence. In general, hardware-based profiling solves many of the problems associated with profiling and makes it a viable technique for use in program optimization.

1.4 Research Contributions

This thesis examines fast and non-intrusive methods for the collection of profile information in hardware. Many commercial microprocessors, such as the Pentium series [32] and the PowerPC 604 [33], incorporate some form of branch handling hardware. Such existing branch handling hardware, along with OS support, can be employed to obtain reliable profile information with imperceptible slowdown (0.4%–4.6%). This allows profiling of an application deployed in the field (e.g., during beta-testing). The application can later be retrieved for profile-based recompilation. The contents of the branch prediction hardware maintain information about the branches in code. This thesis suggests the use of this information for obtaining profiles of the branches in a program. The effectiveness of this kind of information when used in compiler optimizations is also investigated by applying such information to a prominent optimization, namely superblock scheduling. Methods of profiling for both one-level and two-level branch predictors, and results for the quality of information they yield are presented.

Some profiling methods today utilize the hardware support for debugging to provide limited profiling capabilities. For example, the PA-RISC architecture provides for a performance monitoring coprocessor that may be employed for collecting profile data [34]. These techniques require frequent interrupts to be able to gather moderately accurate data. The information collected by such methods is inadequate for the requirements of a profile-driven optimizing compiler. Custom hardware may be used to collect profile information. The *profile buffer*, hardware whose mainline purpose is profiling, is suggested. The profile buffer may be used when branch predictors are absent in a processor, or when the overhead for downloading information from a large predictor table is time-consuming. The profile buffer is far smaller than conventional branch predictor tables, thus enabling the collection of profile information with minimal overhead. Methods to obtain accurate profile information from the profile buffer are also presented. The information obtained from the profile buffer is used in superblock scheduling experiments to verify the accuracy and usefulness of the information.

Statistical theory is used to show that hardware-based profiling yields near-accurate information. Some issues in trace formation using profile information are dealt with. A method of using profile information collected using sampling methods, such as in hardware-based profiling, in trace formation algorithms is suggested. Traces may also be formed by profiling the paths in a program. A hardware structure known as the *history queue* is presented. This structure can be used to profile path traversals

within a program. These paths themselves can then be treated as traces.

Memory disambiguation is another application for profile information. Static analysis of aliasing memory operations during compile time is difficult and time-consuming. Software-based profiling for memory disambiguation can be tedious. The thesis suggests hardware structures for profiling memory operations. Preliminary results are presented for the information that can be obtained from such structures.

Chapter 2

Profiling using Branch Handling Hardware

Profiling in software involves the insertion of counting code in the form of additional instructions at each branch instruction. Such instructions increase the number of instructions to be executed in a basic block. Given that on an average a branch instruction occurs every four to six instructions in integer code, the amount of additional code can slow down the execution of the program considerably. The use of efficient hardware schemes that utilize existing branch handling hardware with OS support to obtain profile information was proposed in Conte, *et al.* [12]. This chapter examines the efficacy of using branch handling hardware to accomplish the task of profiling. The goal is to determine whether the contents of hardware branch buffers can be used to add sufficiently accurate profile information for a given program.

Profile-driven optimizations use a structure known as a *weighted control flow graph* (WCFG), which is a 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 obtained from profile information 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* [2], and the IMPACT project *superblocks* [3]. Chang, *et al.* report a speedup of 15% when superblocks were used to extend the scope of traditional optimizations [3].

Many commercial microprocessors, such as the Pentium series [32] and the PowerPC 604 [33], incorporate some form of branch handling hardware. Such existing branch handling hardware, along with OS support, can be employed to obtain reliable profile information with imperceptible slowdown (0.4%–4.6%). This allows profiling of an application deployed in the field (e.g., during beta-testing). The application can later be retrieved for profile-based recompilation. Since actual usage patterns are profiled, the method also solves the problem of obtaining valid test inputs for profiling. This replaces the *compile-run-recompile* sequence with a less awkward *compile-use-recompile* sequence. In general, hardware-based profiling using branch handling hardware solves many of the problems associated with profiling and makes it a viable technique for use in program optimization.

Most branch handling mechanisms that have been suggested in the literature in

recent times maintain the history of branches preceding a given branch to make a prediction for the direction it will follow [35] [36]. This history can be sampled at regular intervals during the execution of the program to obtain an estimate of the weights for the arcs in a CFG. The sampling of the hardware is performed using kernel-mode instructions for reading branch hardware buffers directly. Several commercial processors implement such instructions. The hardware buffers typically hold target address information along with prediction information. The combination of the target address (the destination of the control flow arc), the buffer tag (the source of the arc), and the prediction information (the arc's weight), fully specify an arc in the WCFG.

Section 2.1 reviews the hardware branch prediction mechanisms used in this chapter. Methods for obtaining profile information from such hardware are discussed in subsections 2.2.1 and 2.2.2. Although these methods are less accurate than full-fledged dynamic profiling using instrumentation code, they are significantly more accurate than static estimates. Metrics to measure the accuracy of the profile information are discussed in Section 2.3. Section 2.4 presents experimental results and discusses the tradeoffs between the various schemes.

2.1 Branch Handling Mechanisms

There are two classes of branch handling mechanisms employed in current processors: *one-level* and *two-level* schemes. One-level schemes use the address of the branch instruction to index into a prediction table (also known as a *branch target buffer*

(BTB) when the target address is also stored with the prediction), which contains a 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 2.1 depicts this process. The most common state machine for one-level schemes is the two-bit counter predictor [37]. This predictor is implemented in several contemporary processors, including the Intel Pentium [32] and the PowerPC 604 [33].

The nominal size for the prediction table in a one-level branch prediction scheme is between 512 and 1024 entries. Experiments show that the two-bit counter, when used in a 1024-entry branch target buffer, achieves a branch prediction accuracy of 90% on-average across the SPECint92 benchmarks. The effects of the size of such a buffer (S) on the number of contentions are examined and plotted in Figure 2.2. The knee of the curves are seen to occur near the 512 to 1024-entry region. The number of contentions is nearly constant for buffer sizes greater than this value. One exception to this is the *gcc* benchmark which has a relatively high number of contentions for all BTB sizes due to its large pool of static branches [38]. For small buffer sizes the prediction error is predominantly influenced by the buffer size. Taking into consideration these observations, a 512 entry branch target buffer is selected for the extent of this study.

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 buffer is indexed using this branch history and stores the state of a predictor. This is depicted in Figure 2.3. These schemes have been studied extensively by Yeh and

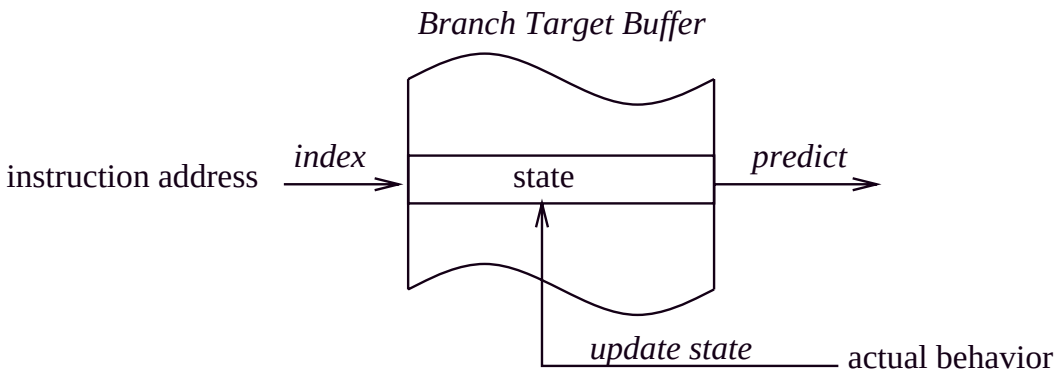

Figure 2.1: One-level branch prediction.

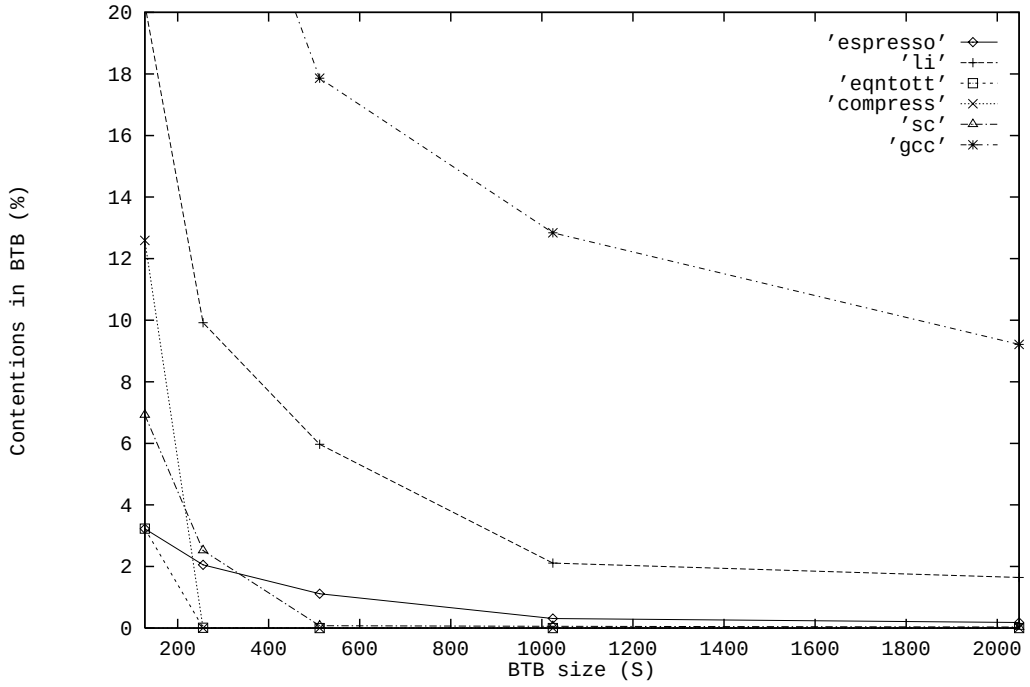

Figure 2.2: SPECint92 benchmarks: Contentions in BTB vs. BTB size.

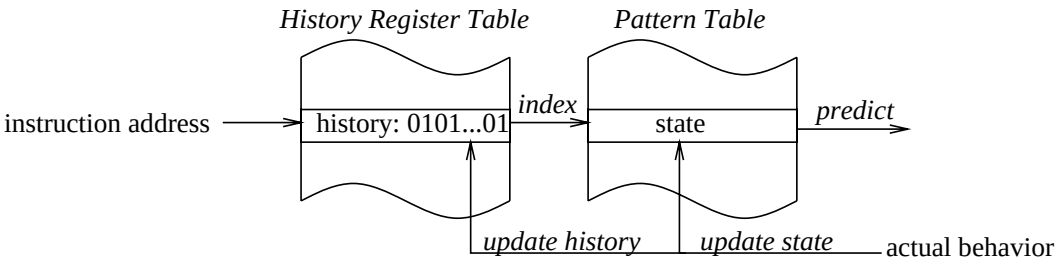

Figure 2.3: Two-level branch prediction.

Patt [35],[36]. Their nomenclature will be used 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. 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 and Patt's algorithm can achieve 96% branch prediction accuracy for SPEC92 benchmarks [35],[36]. The Intel P6 processor employs an adaptation of this two-level scheme.

2.2 The profiling process

The procedure for producing a *weighted control flow graph* (WCFG) via hardware-based profiling using branch handling hardware is as follows:

1. A program is compiled with a special identifier token (magic number), indicating that it contains a table of CFG arcs. This is similar in structure to the table built for arc-based profiling.

2. During execution, the kernel periodically reads the branch handling hardware buffers and uses its contents to increment the arc counters. This is performed at intervals during the execution of the program. Each such instance when information is gleaned from the branch handling buffers is termed a *sampling point*.

3. On program exit, the arc table is updated on disk in a section of the executable file.

The type and quality of profile information depends on the configuration of the branch handling hardware and the sampling process used to obtain the information. These are elaborated upon in the following subsections.

2.2.1 Two-level profiling

The history saved in two-level branch predictors can be used to reconstruct arc weights directly. 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. This is done at each sampling point. The frequency of sampling points and their nature determine the accuracy of the information collected. The following cases are investigated in this thesis:

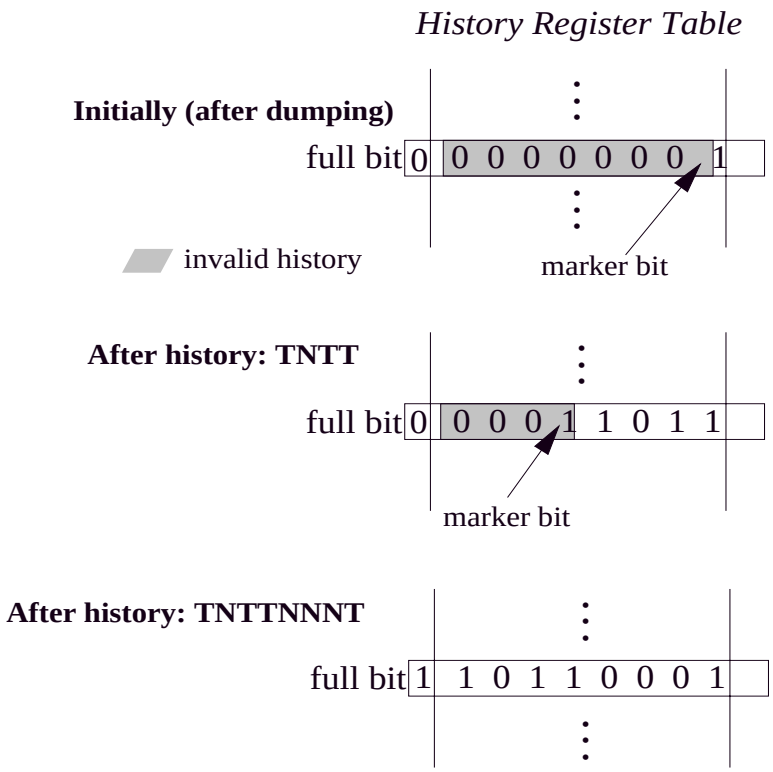

Figure 2.4: The marker bit modification to the two-level scheme for arc weight calculation. This example shows how invalid history is marked as a branch history is updated over time.

- A sampling point occurs at kernel entry points. In this case, the contents of the history register are read and then initialized so that stale information is not read from the register at the next sampling point. The initialization of the history registers to 0 has negligible effect on the prediction accuracy since context switches will normally result in a partial or near-total flush of the buffer. However, entries of '0' in the registers signify not-taken (fall-through) branches. A mechanism is needed to differentiate between '0's due to actual execution and those left over from zeroing the register at the last buffer sampling point.

Several techniques were experimented with in [12], [39]. The best performing was the *marker bit* scheme. This scheme records the boundary between the valid and invalid branch histories, as illustrated in Figure 2.4 [12]. After each sampling point, the history 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 sampling point. Otherwise, only the positions to the right of the leading '1' in the register are valid (i.e., if '$00\cdots01xxxx$', then only the $xxxx$ bits are valid). To complete this scheme, the *full bit* is zeroed at each sampling point. Therefore, only one additional bit per history entry is required, regardless of the length of the HRT entries. The effect of the full bit on the performance of the two-level scheme was found to be negligible [39].

- Alternatively, a sampling point could occur at fixed intervals such as every n instructions. At such a sampling point, the history could be zeroed and a marker bit used to indicate valid history in the history register. However, in this case the zeroing of the history register would degrade the performance of the branch predictor thus making the profiling process obtrusive to the efficient execution of the program. The solution would be to leave the history intact at every sampling point. This would result in history already sampled at the last sampling point being sampled again at the present sampling point. Therefore

such a scheme would sample *stale history* for branches not executed frequently.

2.2.2 One-level profiling

Profiling with one-level schemes requires a mechanism to estimate the arc-weights, since branch histories are not available in such schemes. Although the prediction table in a one-level branch prediction scheme contains a counter, it is a 2-bit counter with an extremely limited range. In addition, the semantics of the update policy for the counter (i.e., increment on taken, decrement on not-taken) further limits its practical range to 1 bit. Instead of using the counter directly, the scheme relies on a count of the total number of branch instructions executed between sampling points. This can be done using on-chip performance monitoring hardware provided in most current processors. Given that B branches were executed between two sampling points, and d entries in the BTB have been accessed during the sample, each entry is assumed to be touched equally $\hat{w} = B/d$ times. Some indicator of the validity of a branch history is required to determine d. This can be implemented without any modification if the BTB maintains a tag store.

The 2-bit counter can be used to enhance the $\hat{w}$ estimate. Branches fall into two categories: those that are heavily biased and those that have time-varying branch histories. Heavily-biased branches tend to saturate the two-bit counter, whereas time-varying branches often (but not always) leave the counter in the middle two states (i.e., weakly-taken or weakly-not-taken). It was assumed for the purposes

of estimation that time-varying branches tend to be taken approximately half as frequently as heavily-biased branches. Then arcs corresponding to heavily-weighted branches are incremented by $(2\hat{w})$, whereas time-varying branches are incremented by $\hat{w}$. This succeeds because of a characteristic of trace selection, where the relative arc weight matters, not the absolute value. The details of the translation are summarized in Table 2.1 [12].

In order to measure the overhead associated with hardware-based profiling, the procedure was prototyped using a Pentium-based AT&T server system [12]. The results are shown in Table 2.2. The results show very little difference in execution time between profiled programs and unmodified programs.

2.3 Comparing profiles

No standard metrics exist for the measurement of error in profiles. One method for comparison is the distribution of average arc weight error versus block weights. The distribution is calculated by computing the maximum differences between the actual and the estimated arc weights for each category of block frequencies. The maximum difference is used in order to avoid over-counting 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.) 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. Defining the maximum

Table 2.1: Approximations used to convert the state of the state machine in the prediction table 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	$2\hat{w}$

Table 2.2: Results for slowdown due to hardware-based profiling. These experiments used a Pentium-based, AT&T 3400-series server system. SPECint92 benchmarks were used.

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%
li	91.4	91.8	0.4%

difference to be,

$$\Delta w_i = \max_{j \in \text{succ}(i)} |w_{ij} - \hat{w}_{ij}| \tag{2.1}$$

the (unnormalized) distribution function is,

$$f_{\text{arcs}}(W) = \sum_{i \text{s.t.} W_i = W} \Delta w_i, \tag{2.2}$$

or the sum of the maximum arc differences for each block with weight W. The average arc error distribution is given by

$$Average\ arc\ error = \frac{\sum_{i \text{s.t.} W_i = W} \Delta w_i}{N_W} \tag{2.3}$$

where N_W is the number of blocks with execution weight W. This metric is an indicator of the accuracy of the arc weights measured using hardware-based profiling methods.

The other metric used measures the effectiveness of hardware-based profiling for a prominent optimization, such as superblock formation or trace selection. 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 [3],[7],[22]. Decisions to include a block in a trace or a superblock are made using profile information. Inaccuracies in the profile information can lead to faulty decisions and consequently performance loss. Either method results in significant code size explosion. To limit this explosion, a threshold is placed on

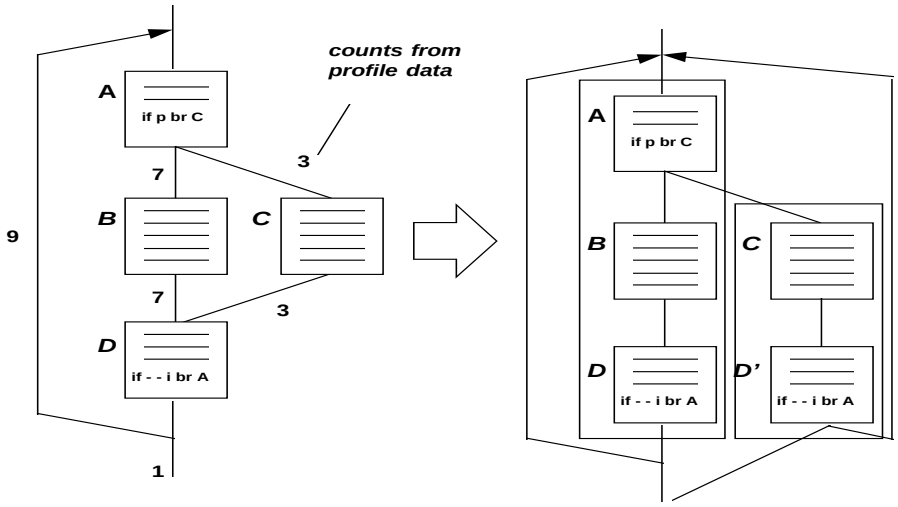

Figure 2.5: Superblock formation using profile information.

the trace selection algorithm. Only blocks with an execution frequency below this threshold are considered for trace membership. The formation of a superblock is therefore highly dependent on the frequency of execution of the basic blocks that it consists of. Figure 2.5 illustrates process of superblock formation.

The accuracy of the profile information is reflected in the parallelism that is obtained as a result of superblock scheduling. Error due to hardware-based profiling may or may not influence the ultimate optimization decisions. The metric of arc error then could be misleading. It may over-estimate or under-estimate the impact of

error. To address this, the profiles obtained from hardware-based profiling are used for superblock scheduling. The resultant speedup over code compiled without profile information is recorded. The speedup is compared with that obtained with perfect profile information.

2.4 Experimental results

The benchmarks enumerated in Table 2.3 from the SPEC92 integer suite were used as the test workloads for the hardware-based profiling experiments. Two specific hardware-based branch predictors were used to collect profile information from the benchmarks: a one-level scheme using a 512-entry buffer (termed *one-level*), and a two-level scheme with a 512-entry history register table containing 10-bit and 16-bit history registers (termed *two-level*). The information obtained from the various profiling methods is utilized to form traces to aid in superblock scheduling. The experiments were performed using the IMPACT architectural framework [40].

Table 2.3: Benchmarks used for evaluation of profiling methods.

Benchmark	Description
compress	file compression utility
espresso	minimization of boolean functions
eqntott	translates boolean equations into truth table
gcc	GNU C compiler
li	C-based LISP interpreter
sc	spreadsheet operations

2.4.1 Two-level Profiling

The performance obtained by scheduling superblocks formed using information from the two 2-level branch predictor profiling methods outlined in subsection 2.2.1 is shown in Figure 2.6. The two-level prediction hardware was sampled on system calls (*2-lev_16_sys*) in which case the history registers were reset after the sampling point. Also shown are the performance results for the *stale history* method at three different interrupt rates, namely, 100,000 (*2-lev_16_1e5*), 1 million (*2-lev_16_1e6*), and 10 million (*2-lev_16_1e7*) instructions. In the *stale history* method the history registers are not reset after the sampling point. Each history register is 16 bits wide. The speedup of superblock scheduling using profile information obtained using these methods over scheduling without profile information is plotted. The speedup is represented as,

$$Speedup = Actual\ speedup - 100.0 \tag{2.4}$$

in this thesis. For instance, if the actual speedup of code using profile information over that without use of profile information is 158%, the speedup is considered to be 58%. These speedups are contrasted with that obtained using perfect profile information (*Exact*). The figure represents the results for code scheduled for an 8 instruction issue machine having two integer units and one each of load, store, and branch units. All units have one-cycle latencies except for the load unit, which has a two-cycle latency.

The performance that profiling yields is demonstrated by the speedup obtained using accurate profile information (*Exact*). This demonstrates the advantages of pro-

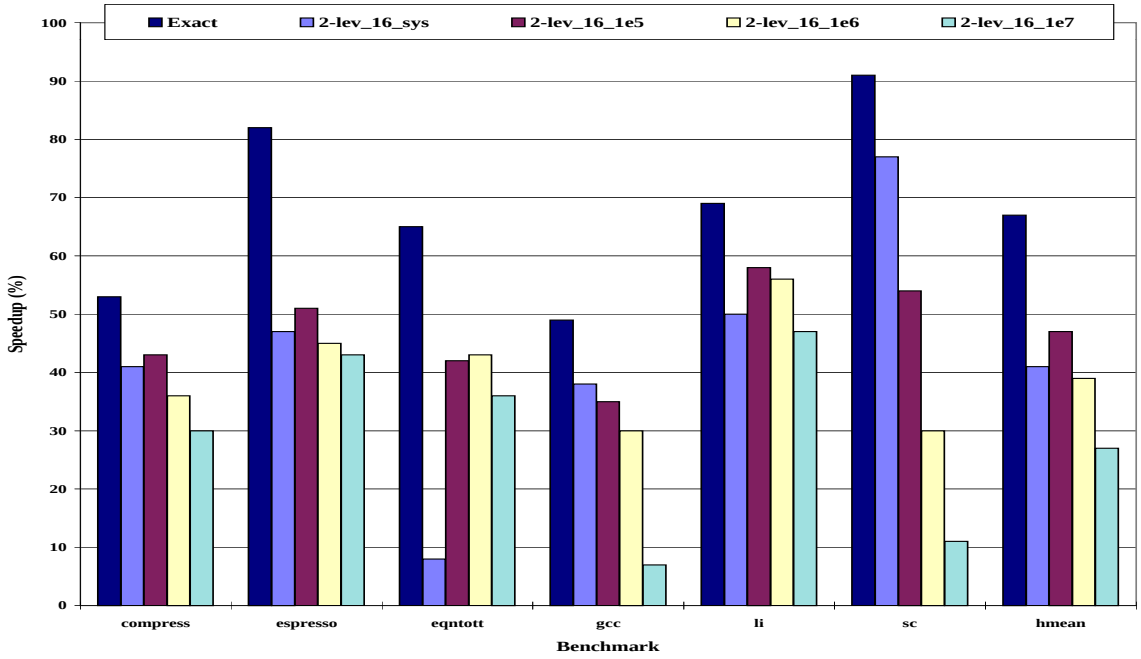

Figure 2.6: Performance with superblock scheduling using profile information from the 2-level branch predictor.

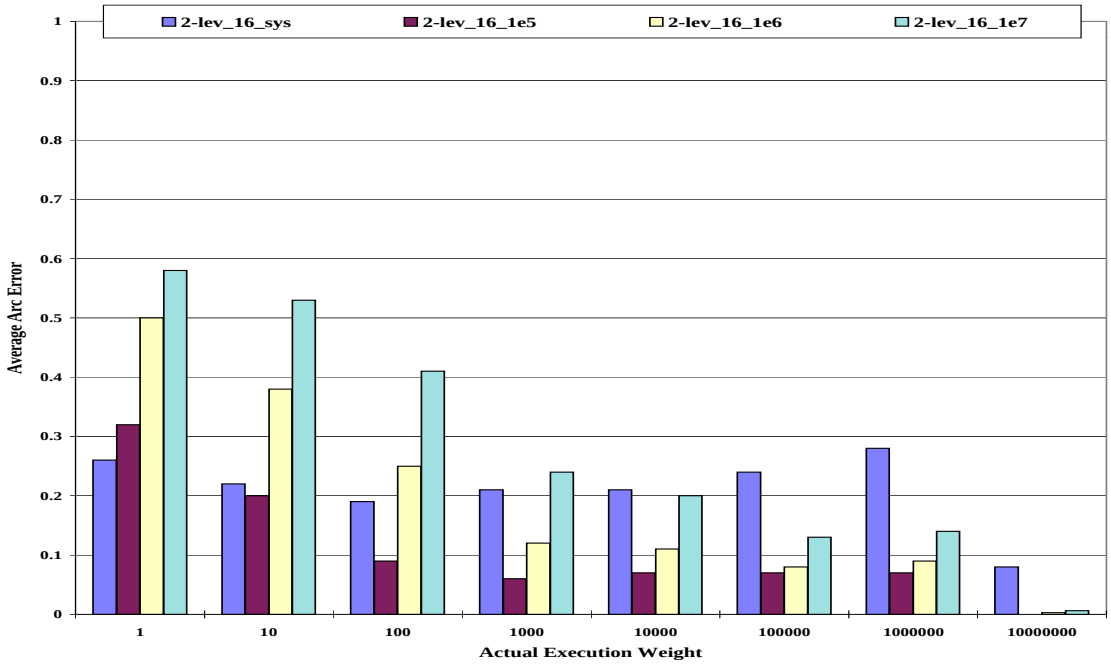

Figure 2.7: Average arc error for profile information extracted from the 2-level branch predictor.

filing itself. The performance for sampling using 2-level branch prediction hardware at a 100,000 instruction sampling interval are greater than *2-lev_16_sys* for almost all benchmarks except *gcc* and *sc*. However, the performance for *sc* using *2-lev_16_sys* is far better than all the other schemes. For *eqntott* the performance for *2-lev_16_sys* is far less than the other sampling intervals. This is due to a lack of system calls or kernel entrances in *eqntott*. This indicates the dependence of sampling at system calls or kernel entrances on the nature of the benchmark being sampled. Sampling at an interval of 1 million instructions yields about the same performance as sampling at 100,000 instruction intervals. However, the performance at sampling intervals of 10 million lags the performance of the other schemes by a large amount. This is especially true for *gcc* which contains a large number of branches. In the case of *sc* the reduction in performance is drastic as the sampling intervals are increased.

Figure 2.7 shows the average arc error for 2-level branch prediction hardware profiling for the various sampling intervals. The average arc error is plotted in categories according to the execution weight of the basic blocks in the benchmarks. The arc error for *2-level_16_sys* is nearly the same across all execution weights. This effect is due to the non-randomness of profiling at system calls or kernel entrances. The sampling points in this scheme are directly related to the location of system calls in the program and therefore the sampling process does not capture branches in proportion to the frequency of their execution. The average arc error for the other sampling methods reduces for blocks with higher execution weights. Sampling at intervals of

a given number of instructions is not directly related to the nature of the code and therefore information for branches more frequently executed tend to be sampled more often. This behavior results in better accuracy for blocks at higher execution frequencies. This behavior also directly impacts performance, since reduced error for these blocks leads to better region formation for these blocks of code and consequently better performance.

The performance for branch history registers with 16 and 10 bits is compared in Figure 2.8. The results for sampling intervals of 100,000 instructions and 10 million instructions are shown. The speedup reduces marginally for *compress*, *espresso* and *sc* for the 100,000 sampling interval. The reduction however is not much since at this sampling interval the number of samples obtained are larger and therefore the loss of information is not significant. The reduction in performance for *li* at a sampling interval of 10 million instructions is greater since at a larger sampling interval every bit of information is important and can contribute to performance. Figure 2.9 shows the arc error plots for the sampling methods using 16 and 10 bits. Most of the arc error can be seen to be at the heavily-executed blocks. In general, the error does not increase much when the number of bits is reduced and thus the impact on performance is minimal.

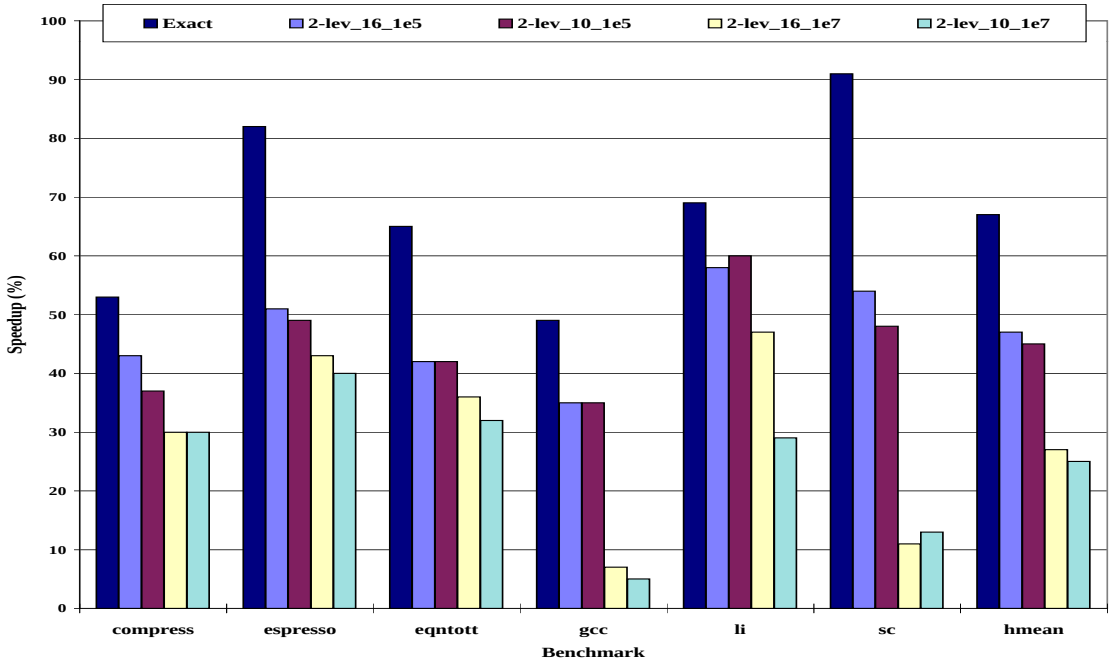

Figure 2.8: Comparison of performance using profile information with 16 and 10 bits in each history register with sampling points at intervals of 100,000 and 10,000,000 instructions.

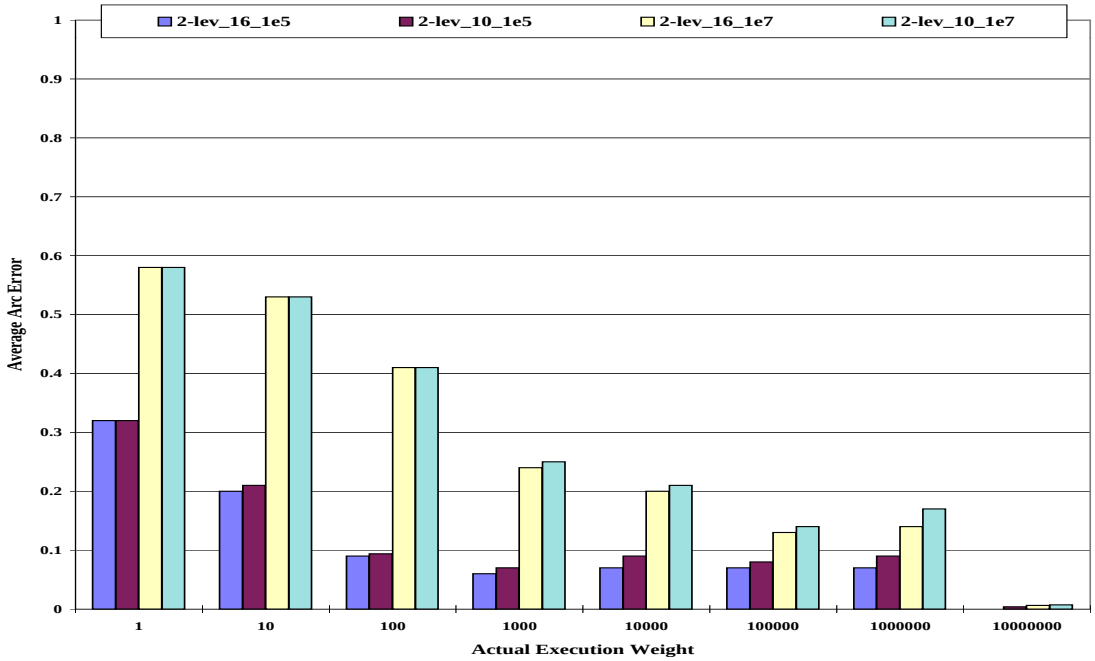

Figure 2.9: Average arc error in profile information for branch history register widths of 16 and 10 in the 2-level branch predictor.

2.4.2 One-level Profiling

Profile information obtained from 1-level branch predictors is an approximation explained in subsection 2.2.2. The performance that this information yields when used to form superblocks is shown in Figure 2.10. As in the 2-level profiling methods, the performance for eqntott suffers when the predictor information is recorded at system call or kernel entry points. The disproportionate number of system calls is the reason. As the sampling interval is increased the error in the profile information collected tends to grow as seen in Figure 2.11 which plots the average arc error for 1-level branch predictor profiling at different sampling intervals. However, the drop in performance as the sampling interval is increased is not as drastic as in the case of 2-level predictor profiling in Figure 2.6. This is because the information obtained from 2-level branch prediction hardware are absolute counts. The limited width of the history registers becomes an important factor at higher sampling intervals. Unlike two-level profiling the information extracted from one-level branch prediction hardware does not depend on how many times a branch was seen. Rather they are approximations based on the branches seen and the state of the state machine for a branch that the was seen. Such approximations seem to work well even at high sampling intervals since the information is not affected by hardware limitations.

The arc error which leads to performance degradation shown in in Figure 2.11 is seen to follow a different pattern than that for 2-level profiling. The error for *1-lev_sys* is higher at frequently executed blocks due to the non-randomness of collecting infor-

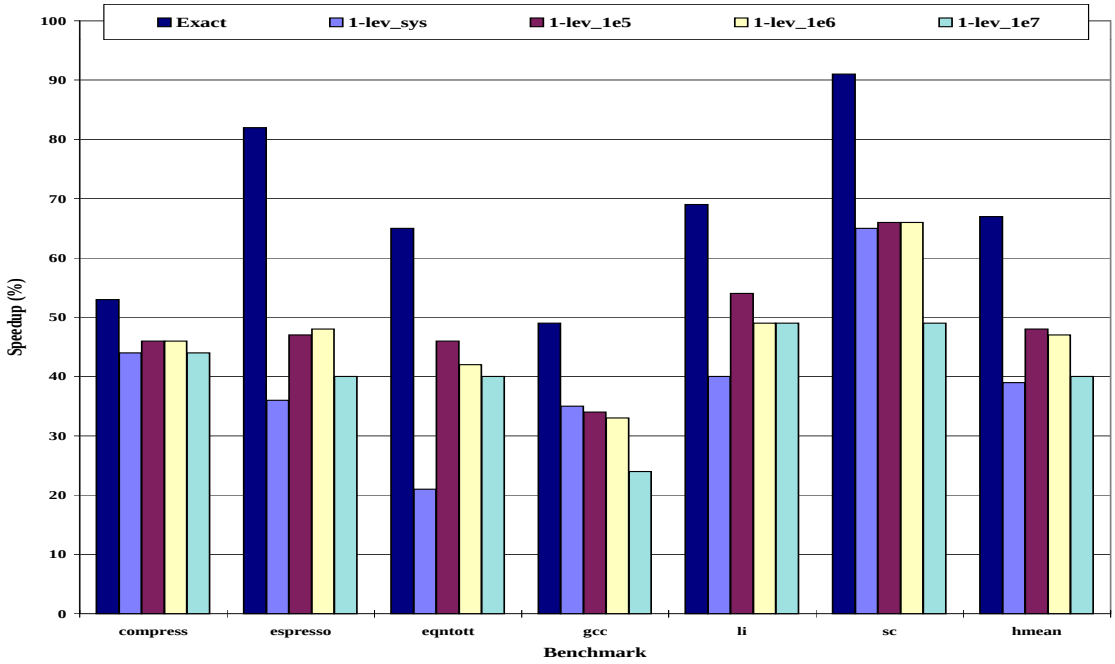

Figure 2.10: Performance with superblock scheduling using profile information from the 1-level branch predictor.

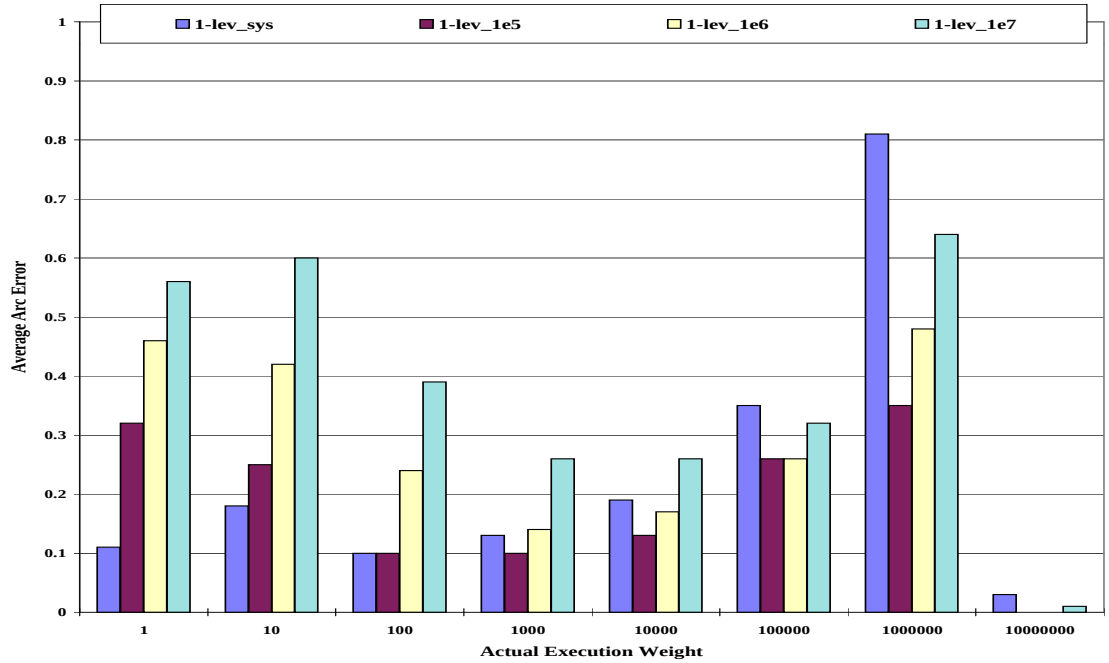

Figure 2.11: Average arc error for profile information extracted from the 1-level branch predictor.

mation at system call points. Also, the presence of system calls and their location in the code affect the quality of profile information collected. The error for the other sampling intervals is high at the more frequently executed blocks and the least frequently executed blocks. It is lower for blocks which are executed about 1000 or 10,000 times. This is again due to the fact that the information obtained is the result of an approximation. Note that the approximation outlined in subsection 2.2.2 rates all branches equally since it only detects if a branch was seen. If so, it computes an approximation based on the state of the state machine for the corresponding branch. Therefore, heavily-executed branches are not given the importance they deserve and the information for such branches is under-approximated. The lightly-executed branches are also given the same weighted priority as the heavily-executed branches. Therefore, the information for these branches tends to be over-approximated. This error on approximation leads to the pattern of error in Figure 2.11.

2.4.3 Comparison of performance between schemes

Figure 2.12 compares the performance after superblock scheduling using information from 1-level and 2-level branch predictions hardware for sampling at system calls and at a sampling rate of 1 million instructions. The performance for *2-lev_16_sys* is generally better than *1-lev_sys* across all benchmarks. One exception is *eqntott*. Due to the lack of executed system calls in *eqntott* little information is obtained due to infrequent sampling points. In such cases, the approximated information used by

the one-level profiling methods is more accurate than the absolute counts obtained using two-level profiling. At a sampling rate of 1 million instructions, both *1-lev_1e6* and *2-lev_16_1e6* have almost the same performance over all benchmarks except *sc*. Again, at a high sampling rate like 1 million instructions, approximations yield better results since they are not limited by the width of the branch history registers like in two-level profiling methods. The harmonic mean of the speedup also indicates the same result.

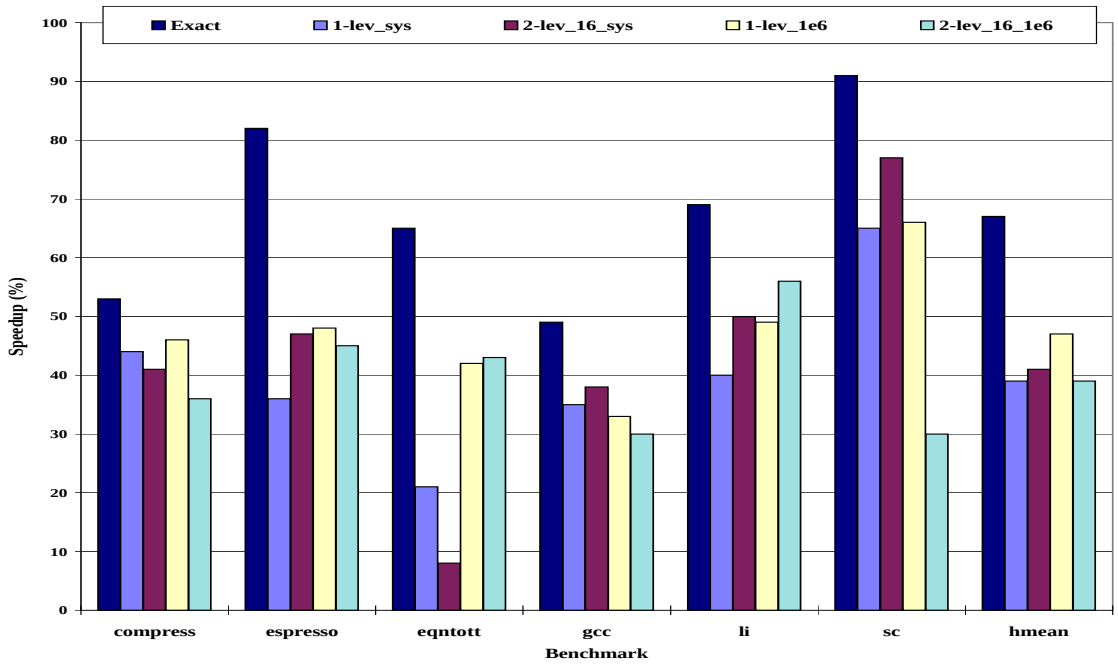

Figure 2.12: Comparison of performance with superblock scheduling using profile information from 2-level and 1-level branch predictors.

2.5 Summary

This chapter explains techniques for extracting profile information from branch prediction hardware. The techniques are described in the context of one-level as well as two-level branch predictors. The efficacy of the profile information obtained using such techniques is measured by applying the information to a prominent profile-based optimization, namely, superblock scheduling. Performance using such profile information approaches that possible by using perfect profile information obtained through software instrumentation.

Chapter 3

Profiling using Custom Hardware

Techniques using branch prediction hardware to collect profile information at kernel entrances were discussed in the last chapter. Such techniques result in a slowdown of 1.02 on average, and 1.05 as a worst case [12]. The negligible slowdown using hardware-based profiling allows software vendors to supply instrumented versions of applications to alpha and beta testers. The profile information based on actual day to day usage can be later retrieved, and final program optimization can be performed. The advantage to hardware based profiling is twofold: It eliminates the need for sample input suites (since actual usage can be captured), and the optimizations are based on actual program usage. Profiling in this manner is commercially appealing because vendors' alpha and beta testing processes are often very well defined, and the hardware-based style of profiling leverages their existing investment to produce better optimized code [11].

Hardware-based profiling using branch prediction hardware as discussed in Chapter 2 requires the presence of appropriate branch prediction hardware in the processor. Such hardware may not always be available. Limited information is contained in each entry of the branch prediction hardware. The performance gain seen in using such hardware in the Chapter 2 is not much affected by contentions in the branch prediction hardware due to the large size of the buffers used. Smaller sizes of the branch prediction hardware could lead to contentions resulting in worse profile information, and consequently little performance gain on using the profile information for program optimization. These concerns are motivation for investigating special-purpose hardware with the mainline purpose of collecting profile information. Some profiling methods today utilize the hardware support for debugging to provide limited profiling capabilities. For example, the PA-RISC architecture provides for a performance monitoring coprocessor that may be employed for collecting profile data [34]. One other technique used is *program counter sampling*. Profiling using these techniques requires frequent interrupts to be able to gather moderately accurate data. The information collected by such methods may also be inadequate for the requirements of a profile-driven optimizing compiler due to their coarse-grained nature. For example, information obtained using *program counter sampling* may be effectively used to implement a *code-layout* optimization, but would not be able to contribute a great deal to optimizations such as *scheduling* which require fine-grain profile information.

This chapter proposes the *profile buffer* for the collection of profile information.

The profile buffer is a special purpose buffer used for the mainline purpose of profiling. It is similar in slowdown to branch predictor based profiling. The profile buffer consists of counters which accumulate the frequencies of the branches in code. The sizes of the profile buffer investigated in this chapter are small in comparison to the sizes of branch prediction hardware. However, it fulfills the profiling demands of an optimizing compiler by providing high accuracy. In order to evaluate its effectiveness, the profile information obtained from this method is applied to a prominent optimization, namely superblock scheduling [3]. The performance measurement data based on the execution of the optimized program indicates that a small buffer (32–64 entries) can obtain a high degree of accuracy for a relatively small amount of slowdown (1.02 times).

The design tradeoffs for the profile buffer are examined in this chapter. The *profile buffer* and its implications in terms of hardware, the instruction set architecture (ISA) and the operating system are discussed. Modified schemes of employing such a buffer are discussed in sections 3.2.1–3.2.3. In particular, three methods for profile buffer indexing, *address-mapping*, *selective indexing*, and *compiler indexing*, are presented. The *address mapping* scheme employs a hashing algorithm to map addresses into the profile buffer. The other techniques, *selective indexing* and *compiler indexing*, use minor extensions to the ISA. In the case of *selective indexing*, an additional bit is added to the branch instruction. However, the *compiler indexing* method uses an additional field for the encoding of the profile buffer index. In addition, the profile

buffer with associativity is examined and results presented. Associativity is seen to help greatly in obtaining accurate profile information using the profile buffer.

3.1 The Profile Buffer

The *profile buffer* is a set of counters that extracts information from the reorder buffer in the processor. The results of resolved branches in the reorder buffer are used to update the count in the appropriate entry in the profile buffer. These counts are then written to memory at context-switch time or on a timer interrupt. Subsection 3.1.1 describes a node-based profile buffer and subsection 3.1.2 describes an arc-based profile buffer.

3.1.1 Node-based profile buffer

The *node-based profile buffer* is shown in Figure 3.1. It consists of a set of counters, with each counter counting the number of times a branch was seen. A resolved branch indexes into the buffer using the address of the branch. The count in the entry is incremented. The contents of the buffer are written to memory either on a context switch or a timer generated interrupt which constitutes a sampling point. The use of the node-based profile buffer, yields a node-based profile i.e. frequencies of each basic block in the program are known. Profiling in this manner is comparable to *program counter sampling* which also yields a node-based profile. Such a profile may be used for compiler optimizations such as *code layout*. However, most compiler optimizations

like superblock scheduling require finer-grain information which provide the frequencies of arcs between basic blocks. While an arc-based profile can be operated on to generate a node-based profile, the converse is not true. Knowledge of the frequencies of nodes in a control flow graph may not always allow inference of the frequencies of the arcs in the graph. One example is shown in Figure 3.2.

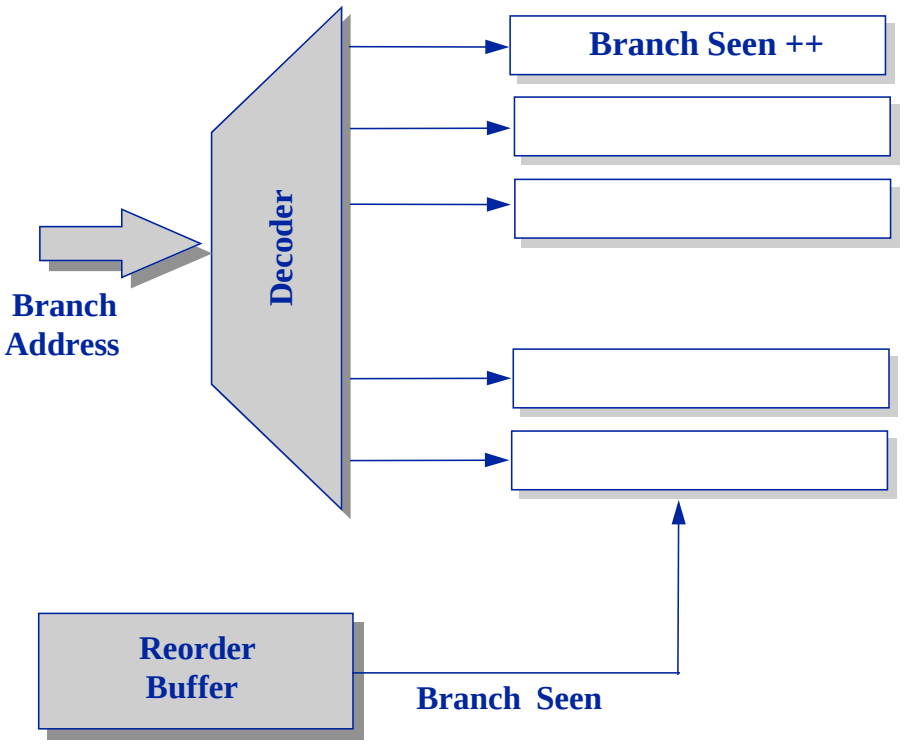

Figure 3.1: Node-based profile buffer. (Entries in the buffer are counters).

3.1.2 Arc-based profile buffer

An arc-based profile coupled with a node-based profile can satisfy the needs of most control flow compiler optimizations. An arc-based profile provides adequate informa-

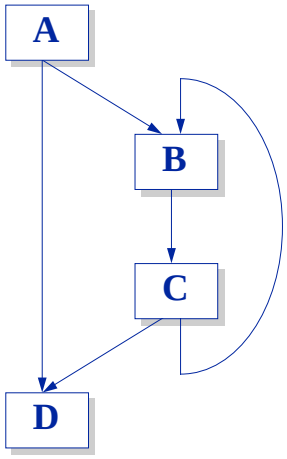

Figure 3.2: Example control-flow graph for which arc weights cannot be deduced from knowledge of node weights.

tion for the generation of a node-based profile. To obtain a near-accurate arc-based profile while maintaining a low profiling overhead, the *arc-based profile buffer* is proposed. This buffer is illustrated in Figure 3.3. The profile buffer is composed of a set of counters, with each entry in the buffer consisting of two fields: the number of times the branch is taken (*num_taken*) and the number of times a branch is not-taken or falls-through (*num_not_taken*). The profile buffer works in close cooperation with a standard reorder buffer. On instruction completion, the reorder buffer entry for a branch contains the branch address and the information as to whether the branch was taken or not-taken. At write-back, when the branch retires, the profile buffer is referenced using the branch address or calculated index and one of the two fields of the corresponding profile buffer entry is updated based on the result of the branch.

The information in the *profile buffer* needs to be stored back to memory periodically. The storage of the buffer information may take place on a context-switch.

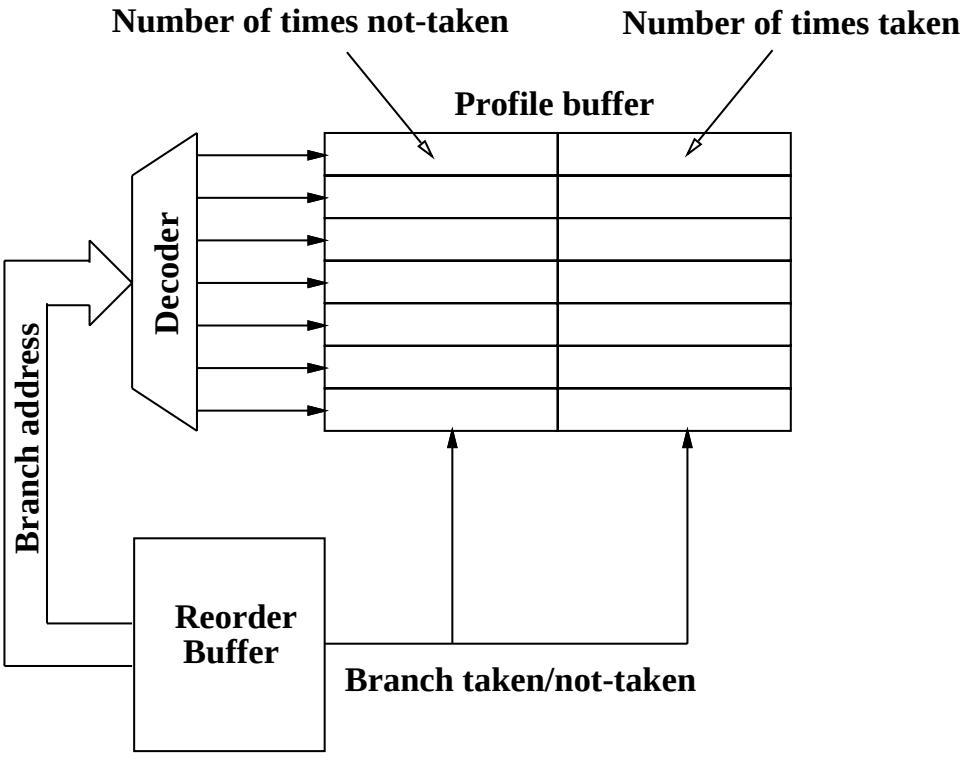

Figure 3.3: The *arc-based* profile buffer. (Entries in the buffer are counters).

Alternatively, the operating system may generate a timer interrupt and perform the required action. The operating system controls the storage of the profile buffer information to memory. The update of the profile buffer is not in the critical path of the processor since it is done after a branch is resolved and therefore does not cause the processor to stall. Additionally, the hardware proposed does not impact the cycle time because the actions associated with the *profile buffer* constitute a write-only, post-completion phase, where the branch is allowed to update the counters in the buffer over a multiple number of cycles.

Since the updating of the profile buffer is entirely done by the processor off the critical path the only slowdown possible is due to the operating system interrupt and the storage back to memory. Branches that have been most recently accessed are likely to be accessed again in the near future (i.e., they have temporal locality). Therefore, the interrupts may be spaced well apart without degrading the information contained in the *profile buffer*, thereby minimally affecting the execution speed of the program. Experiments to measure slowdown using branch–handling hardware and dumping out its contents on a context–switch were conducted in [12]. The authors reported a maximum slowdown of 1.046 times for the SPECint92 benchmarks, with an average slowdown of 1.02 times on a Pentium-based AT&T server system when the size of the branch hardware employed in the experiments was a 512-entry buffer. A large buffer was required to provide for the many branches in each of the SPECint92 benchmarks studied [12]. The profile buffer information is recorded in a manner similar to branch

handling hardware. The size of the profile buffer being much smaller (e.g., 32 entries) it is expected to have a slowdown comparable to or lower than the average of 1.02 times reported in [12].

The profile buffer must be as small as possible to minimize the cost of additional hardware. As a consequence of this, a design space ranging from eight to 64 entries in the buffer is considered, where each entry is divided into two 16-bit counters. The profiling information obtained in this manner may not be accurate because the buffer size may be too small to handle the large number of branches that may exist in a program. Given a restricted design space such as in this chapter, the reduction of contentions is highly relevant to the profile buffer indexing scheme. Contentions cause the inaccuracies in the profiling information obtained during program execution. The distribution is calculated by computing the maximum differences between the actual and the estimated arc weights for each category of block frequencies. The maximum difference is used in order to avoid over-counting 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.) 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. Defining the maximum difference to be,

$$\Delta w_i = \max_{j \in \mathrm{succ}(i)} |w_{ij} - \hat{w}_{ij}| \tag{3.1}$$

the (unnormalized) distribution function is,

$$f_{\text{arcs}}(W) = \sum_{i \text{s.t.} W_i = W} \Delta w_i, \tag{3.2}$$

or the sum of the maximum arc differences for each block with weight W. The average arc error distribution is given by

$$Average\ arc\ error = \frac{\sum_{i \text{s.t.} W_i = W} \Delta w_i}{N_W} \tag{3.3}$$

where N_W is the number of blocks with execution weight W. This metric is an indicator of the accuracy of the arc weights measured using hardware-based profiling methods.

The SPECint92 benchmarks were used to evaluate performance. Each program was divided into a collection of blocks of instructions in which control may only enter at the top but may leave at one or more exit points (better known as *superblocks* [7]). The profile information collected using the profile buffer scheme was applied to a superblock formation algorithm implemented in the University of Illinois IMPACT compiler [40],[23]. The accuracy of the profile information is reflected in the parallelism that is obtained as a result of superblock scheduling. All experiments were simulated on an eight–issue machine with having two integer units and one each of load, store, and branch units. All units have one-cycle latencies except for the load unit, which has a two-cycle latency. The resultant speedup over code compiled

without profile information is noted. Note that speedup is computed as,

$$Speedup = Actual\ speedup - 100.0 \tag{3.4}$$

for the purposes of this thesis. For instance, if the actual speedup is 158%, the speedup is represented as 58%. The speedup is compared with that obtained with perfect profile information.

3.2 Indexing Schemes

The following three subsections outline three different indexing schemes. The performance of the profile buffer at different sizes using these indexing schemes is presented. The three indexing schemes that were reviewed are, address mapping, selective indexing, and compiler indexing.

3.2.1 Address Mapping

The first approach to indexing the profile buffer is *address mapping*. This simple method uses the branch instruction address to reference the profile buffer. It is identical to indexing schemes used in branch prediction hardware, and its performance is comparable to that presented in [12],[41]. No complicated optimization of access schemes are necessary because every branch instruction is used in *address mapping*. The results for this scheme are presented in Table 3.4. A distinct performance increase

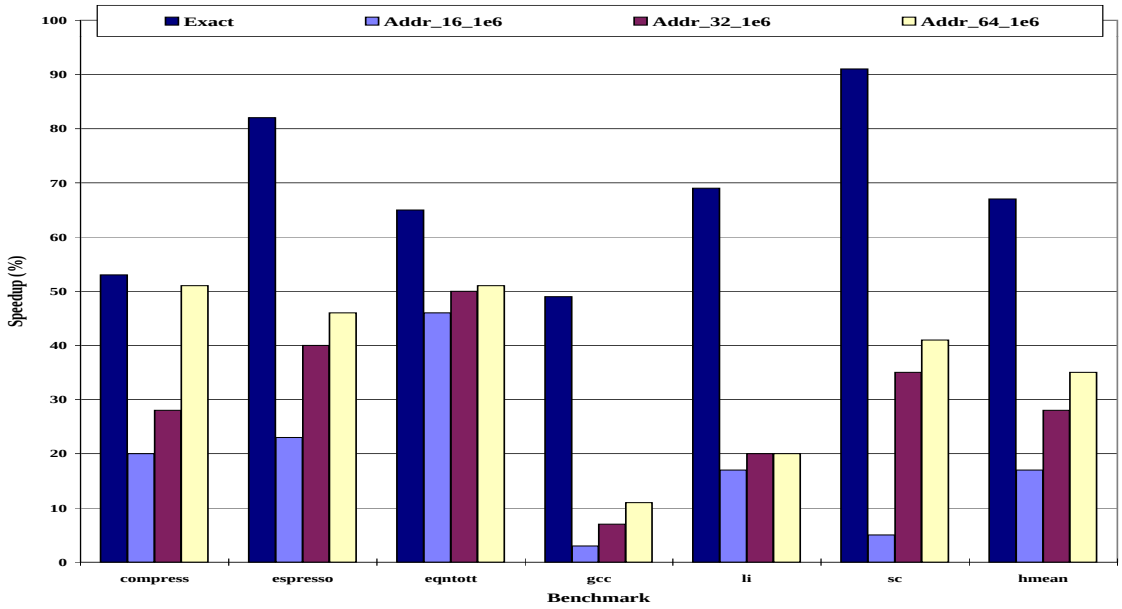

Figure 3.4: Speedup for superblock scheduling using profile information from the profile buffer with address mapping. (*Addr_n_i* indicates a profile buffer with n entries sampled at intervals of i instructions.)

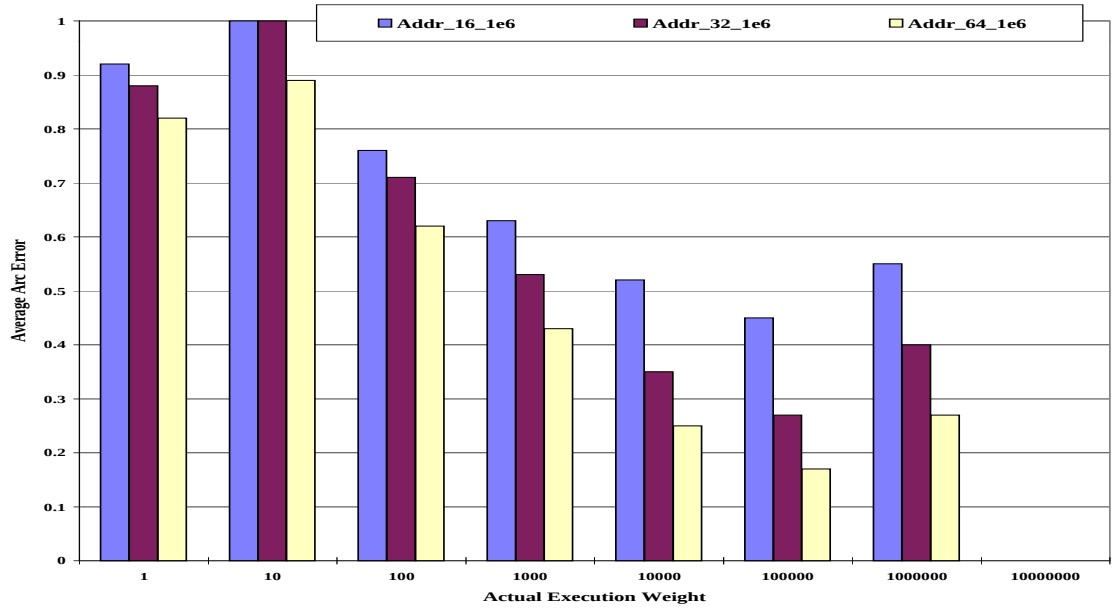

Figure 3.5: Average arc error in profile information from profile buffer with address mapping.

Table 3.1: Contentions as percentage of total accesses for *address mapping*. (Intra-procedural contentions)

Benchmark	Percent Contentions			
	8	16	32	64
compress	36.50%	30.05%	19.84%	6.62%
espresso	45.73%	22.49%	9.76%	3.71%
eqntott	36.17%	1.44%	0.48%	0.04%
gcc	40.35%	26.95%	17.68%	10.82%
li	47.61%	35.25%	23.61%	11.97%
sc	65.32%	41.03%	19.93%	12.47%
mean	45.28%	26.20%	15.22%	7.61%

Table 3.2: Contentions as percentage of total accesses for *address mapping*. (Inter-procedural contentions)

Benchmark	Percent Contentions			
	8	16	32	64
compress	0.00%	0.00%	0.00%	0.00%
espresso	2.35%	2.01%	1.77%	1.27%
eqntott	2.55%	0.25%	0.15%	0.10%
gcc	23.69%	28.79%	30.82%	29.87%
li	23.51%	26.72%	27.13%	25.99%
sc	11.23%	12.15%	10.56%	8.65%
mean	10.56%	11.65%	11.74%	10.98%

is seen as the size of the profile buffer is increased. The harmonic mean (*hmean*) indicates that the performance does not compare well to *Exact*. The *average arc error* is shown in Figure 3.5. It may be observed that the arc error is extremely high, leading to the performance degradation. The *arc error* reduces as the size of the buffer is increased. At 64 entries (*Addr_64_1e6*), the speedup approaches that possible with full profile information (*Exact*) for *compress*. However, for most of the other benchmarks the performance levels achieved fall short of *Exact*. The error in the measurement of arc weights using *address mapping* is due to the large number of contentions in the profile buffer when simple *address mapping* is employed. The contentions for entries in the profile buffer are shown in Tables 3.1 and 3.2. The contentions are split into two categories: *intra-procedural* and *inter-procedural*. Intra-procedural contentions happen when two branch instructions from within the same function map to the same entry. Branch instructions from different functions mapping to the same entry are the cause of inter-procedural contentions. It is evident that the number of contentions are large over all the benchmarks. In fact, the benchmark *li* has almost three contentions per every four profile buffer accesses for an eight-entry buffer. These results indicate that reducing the number of contentions should be a priority for obtaining better profile information with the profile buffer. Traditionally, branch target buffers (BTB) that have similar properties as the profile buffer have overcome the problem by employing larger buffers (512 or 1024 entries) [32], [42], but this implies added hardware cost. An alternative solution is to reduce the number of branches

that access the buffer. This alternative is explored in the next two subsections.

3.2.2 Selective indexing

The number of contentions in the profile buffer can be reduced by reducing the number of accesses to the buffer. Reducing the number of accesses to the buffer can be expected to greatly reduce the number of contentions, and consequently the error in the profile information. Several algorithms have been suggested to reduce overhead in the code instrumentation methods of profiling [29], [31], [30]. This subsection describes two methods to reduce the number of branches that access the profile buffer. One for a *node-based* profile using the node-based profile buffer. The other for an *arc-based* profile using the arc-based profile buffer.

Reduction of access in a node-based profile buffer

The *selective indexing* profiling scheme for the node-based profile buffer uses the Knuth-Stevenson algorithm to determine the branches that should access the profile buffer. Knuth and Stevenson [31] describe a method for reduction in the number of nodes that need to be probed. The algorithm uses a graph transformation followed by the spanning tree algorithm to produce a minimal set of basic blocks whose frequencies need to be measured. Given a CFG with V basic blocks and E edges, and there exists a vertex c and arcs $c \rightarrow a$ and $c \rightarrow b$, then basic block a is considered to be in the same equivalence class as basic block b ($a \equiv b$). The resultant transformed graph consists

of nodes that correspond to the equivalence classes and edges that correspond to the nodes in the original graph.

Given the resultant transformed graph with n vertices and m edges, the frequencies of only $m - n + 1$ edges need to be measured. The frequencies of the other edges can then be computed from the measured frequencies. An application of the spanning tree algorithm results in a tree whose edges represent the nodes that do not need to be probed. The nodes that represent the arcs absent in the spanning tree form the set of nodes whose frequency is to be measured.

Figure 3.6 shows an example CFG with basic blocks $N_1, N_2, \ldots, N_6$ and the edges between them. The graph transformation yields equivalence classes and the edges between them which correspond to the nodes in the original CFG. The application of the spanning tree algorithm to find the minimal set of nodes to be probed assumes that the CFG is entered only once. However, a function may be invoked more than once and therefore a pseudo arc N_6 is inserted to represent node N_6. The spanning tree algorithm when applied to the reduced graph leaves out edges N_3, N_5 and N_6 which represent the nodes that need to be probed. It may be observed that knowledge of the frequencies of these nodes allows the frequencies of all the other nodes to be deduced.

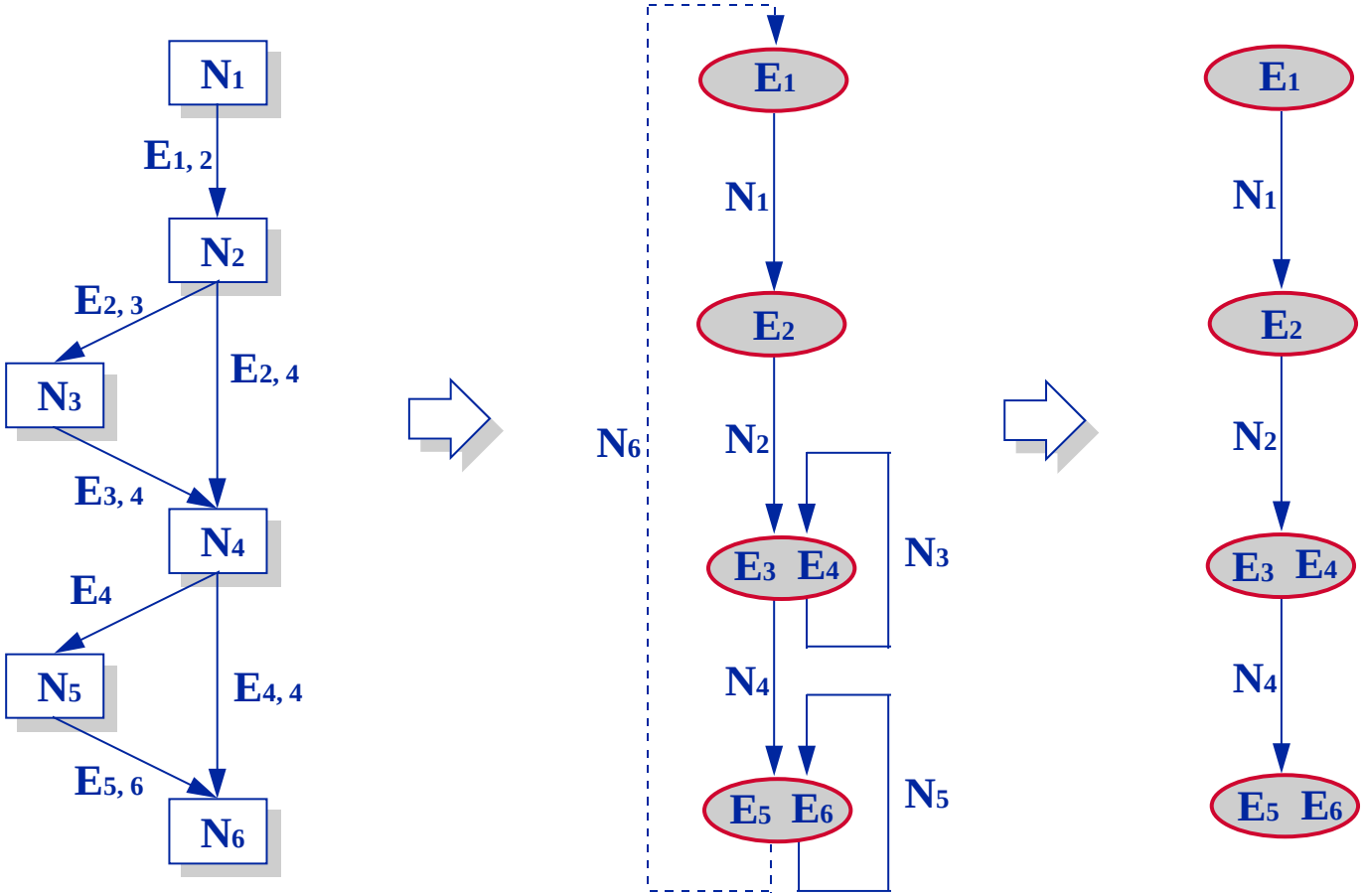

Figure 3.6: The Knuth-Stevenson Algorithm to determine a minimal set of nodes to profile.

Reduction of accesses in the arc-based profile buffer

Reduction of accesses in the arc-based profile buffer is possible by applying the spanning tree algorithm to the control flow graph. Figure 3.7 illustrates the process of obtaining a minimum set of arcs to probe in a given control flow graph. Figure 3.7(a) shows a control flow graph with 6 nodes and 8 edges. A pseudo-edge has been added from the exit block to the start basic block to model the entry count of the CFG. Application of the spanning tree algorithm should yield $8 - 6 + 1 = 3$ edges whose frequencies need to be measured. Figure 3.7(b) shows the CFG with the edges in the spanning tree marked with thick lines. The edges not in the spanning tree are the edges that need to be probed, namely, edges $E_{3,4}$, $E_{4,6}$ and $E_{5,6}$. The arc traversals for these arcs would have to be measured if software instrumentation was employed. However, when using hardware-based profiling as described in this thesis, the branch instruction is the instrument of profiling. The branch instructions corresponding to the arcs requiring measurement update the profile buffer. Therefore, for the purposes of hardware-based profiling the branch instructions in nodes N_3, N_4 and N_5 access the profile buffer.

The arc-based profile buffer has two counters, one for *branch taken*, the other for *branch not-taken*. However, in the case of fall-thru branches, i.e. nodes with a single outgoing edge, only one counter would be used. This leads to under-utilization of the counters in the profile buffer. Also, nodes with more than two outgoing edges can cause inaccuracy in the profile information obtained since one counter is shared

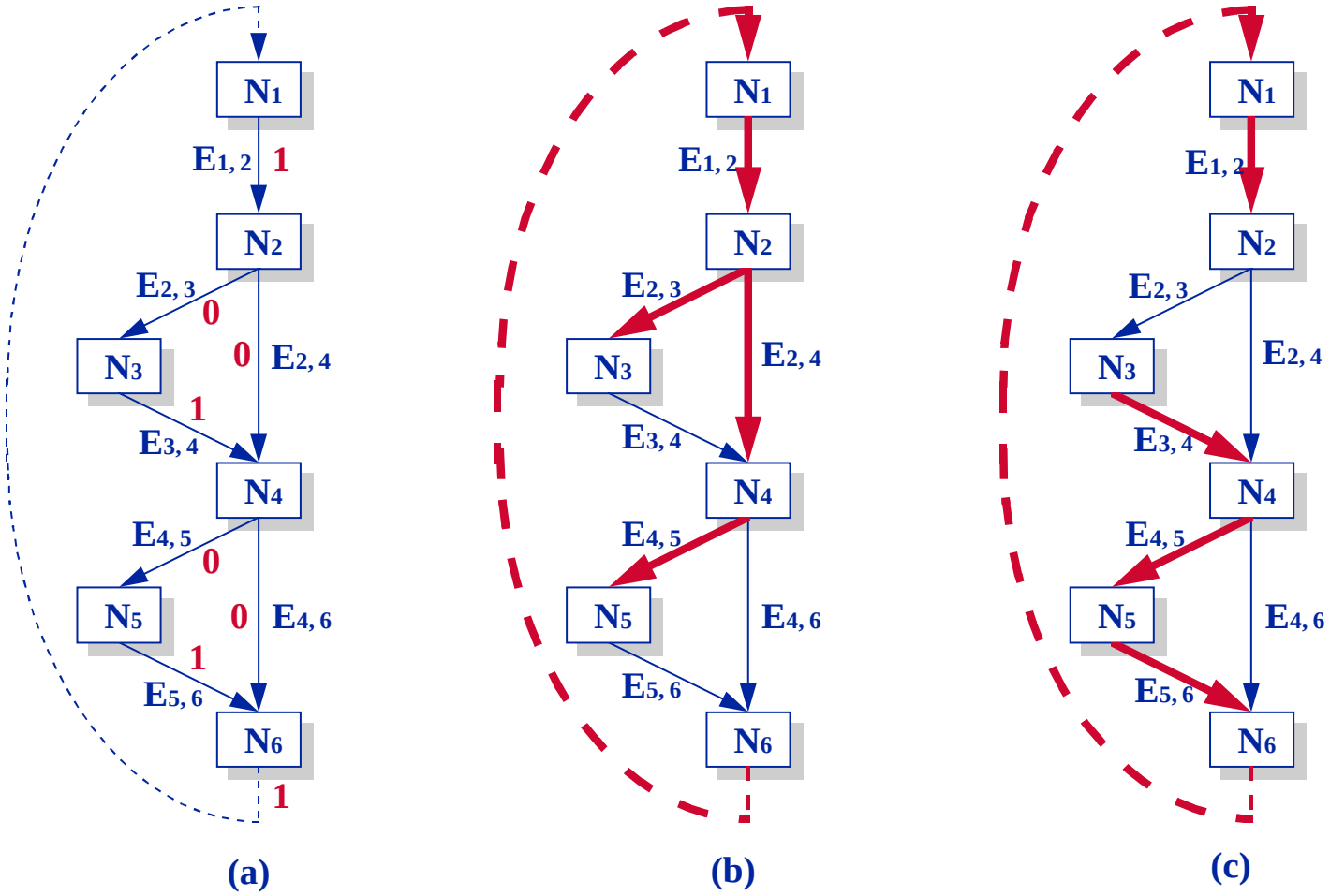

Figure 3.7: The spanning tree algorithm to determine a minimal set of arcs to profile.

by multiple edges. Switch statements lead to such instances. Switch statements may be converted to multiple *if* statements, to avoid this problem. Alternatively, a cost assignment on the graph could be used to overcome both problems mentioned above. Figure 3.7(a) shows weights assigned to the arcs in the control flow graph. All outgoing edges out of a block are weighted equally, with the number of arcs out of the block. For example, a block with three outgoing edges would result in each outgoing edge being assigned a weight of 3. However, in the case of blocks with one or two outgoing edges weight assignment is reversed. The weight assigned to an edge out of a block which has two outgoing edges is 1, while that for an edge out of a block with a single outgoing edge is 2. The application of a maximal spanning tree algorithm to such a weighted control flow graph results in the arcs with the higher weights being included in the spanning tree. Therefore, blocks with more than or less than two outgoing edges would be included in the spanning tree. Since only the complement of the arcs in the spanning tree need to be probed, the inclusion of such arcs in the spanning tree can be partially avoided. The resulting graph after the application of the maximal spanning tree algorithm is shown in Figure 3.7(c). The thick lines indicate the arcs that form the spanning tree. The arcs that need to be probed are $E_{2,3}$, $E_{2,4}$ and $E_{4,6}$. For hardware-based profiling, the branch instructions in nodes N_2 and N_4 need to access the profile buffer. Notice that with the use of the cost assignment and maximal spanning tree algorithm not only is the accuracy of profile information increased by avoiding nodes with more than two branches, the

number of branch instructions that access the profile buffer is also decreased from three without the cost assignment, to two for the example shown in Figure 3.7.

Requirements for Selective indexing

This chapter only considers the arc-based profile buffer. The arc-based profile buffer counts the frequencies of arcs. This is achieved by counting the number of a times a branch is *taken* or *not-taken*. In order to distinguish the branches that need to be profiled from those that should not be profiled, a minor modification is required in the instruction encoding of a branch. For this investigation, an additional bit is added to indicate whether or not the instruction needs to update information in the *profile buffer*.

There is an alternative to changing the existing ISA if the ISA provides for complements of every branch condition (for example, branch equal (BEQ) and branch not equal (BNE)). One half of these branch conditions can be intercepted by the *profile buffer* while the complements are ignored. For example, a BEQ operation could update the buffer while the BNE operation would not. If an operation is BNE and it's information needs to be recorded, this condition could be inverted and the recording operation changed to BEQ during code generation. Another alternative for architectures that use predicated branches (such as *HPL PlayDoh* [43]), is to profile every branch predicated on even-numbered predicates, but not profile branches predicated on odd-numbered ones. Thus, selective indexing can be retrofitted into an existing

ISA without modification to the encoding.

Results for Selective indexing

The results for the *selective indexing* scheme using the suggested ISA change are presented in Tables 3.3 and 3.4. The percent change in accesses is computed against the results produced by *address mapping*. The percent change in the number of accesses is considerable for all benchmarks. In particular, the number of accesses for four of the benchmarks are reduced by more than 30% with reductions up to 60.25% for *compress* and 45.17% for *eqntott*. *Selective indexing* affects the number of intra–procedural as well as inter–procedural contentions across all buffer sizes. The percentage of intra-procedural contentions for *gcc* for a 32-entry buffer is 11.52% compared with 17.68% for *address mapping*. For 64 entries, the percentage of intra–procedural contentions for *gcc* is reduced to 7.29% from 10.82%. Similar reductions in intra–procedural contentions are also seen for *li* and *sc*. A drastic reduction in the inter–procedural contentions is also seen, especially for *gcc*, *li*, and *sc*. The decrease in inter–procedural contentions is more than 10% for *gcc* and *li* for 32-entry and 64-entry profile buffers. However, the percentage of inter–procedural contentions is still high, in the range of 10%–20% for *gcc* and *li* for a 64-entry buffer. This high percentage of inter–procedural contentions could lead to considerable error in the profile information. However, the reductions in the intra-procedural contentions may be expected to provide better performance than that obtained using *address mapping*.

Table 3.3: Effect of reduction in profiling branches using *selective indexing*. (Intra-procedural contentions). (Contentions are shown as percentage of original accesses).

Benchmark	Change in accesses	Percent Contentions			
		8	16	32	64
compress	-60.25%	14.80%	9.01%	0.00%	0.00%
espresso	-30.13%	19.49%	10.81%	5.24%	1.03%
eqntott	-45.17%	0.56%	0.95%	0.06%	0.00%
gcc	-29.84%	25.25%	16.98%	11.52%	7.29%
li	-36.31%	24.24%	18.59%	11.33%	6.49%
sc	-19.77%	43.22%	23.19%	6.85%	2.08%
mean	-36.91%	21.26%	13.26%	5.83%	2.82%

Table 3.4: Effect of reduction in profiling branches using *selective indexing*. (Inter-procedural contentions). (Contentions are shown as percentage of original accesses).

Benchmark	Percent Contentions			
	8	16	32	64
compress	0.00%	0.00%	0.00%	0.00%
espresso	1.08%	0.63%	0.55%	0.45%
eqntott	1.53%	0.08%	0.07%	0.06%
gcc	17.27%	19.67%	19.69%	18.30%
li	15.63%	17.68%	16.98%	13.66%
sc	6.78%	6.99%	4.91%	3.95%
mean	7.05%	7.51%	7.03%	6.07%

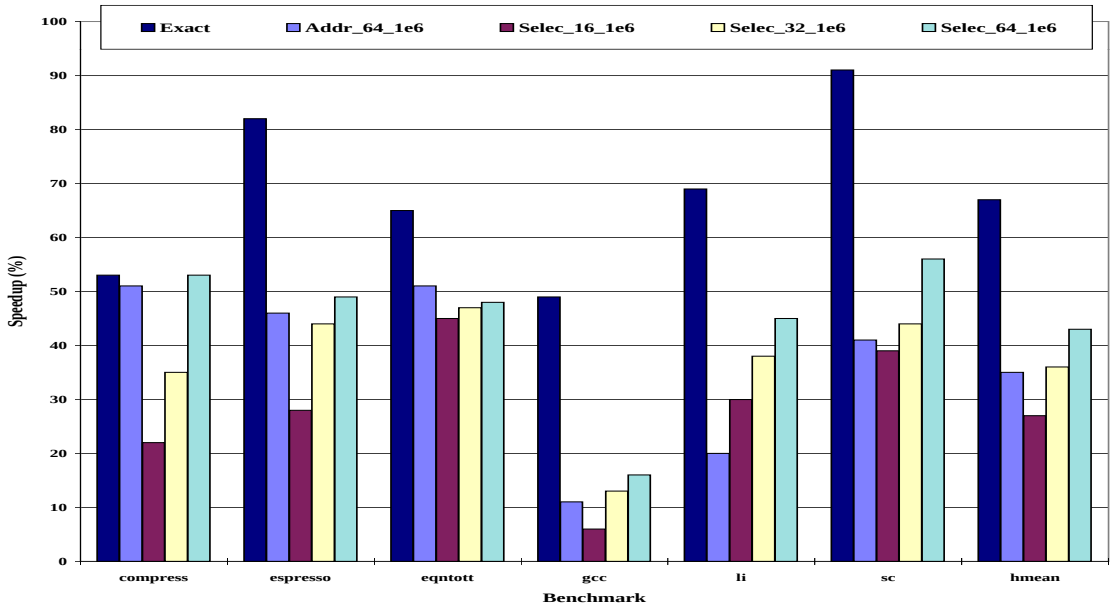

Figure 3.8: Speedup for superblock scheduling using profile information from the profile buffer with selective indexing. (*Selec_n_i* indicates a profile buffer with n entries sampled at intervals of i instructions.)

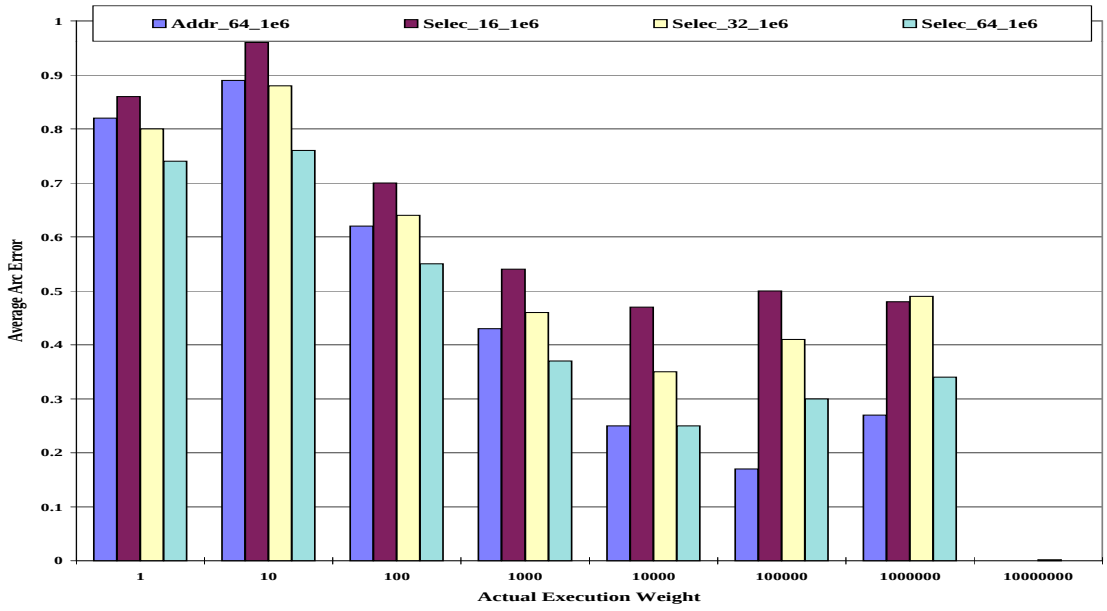

Figure 3.9: Average arc error in profile information from profile buffer with selective indexing.

Figure 3.9 compares the *average arc error* for the *selective indexing* technique to that for a 64-entry buffer using *address mapping*. The error for lightly and medium executed branches, i.e. branches executed between 10^0 and 10^4 times, the arc error is reduced compared to the address mapping technique for the 64–entry buffer. The error for basic blocks executed between 10–99 times is less by at least 10%. The 16-entry and 32-entry buffers using selective indexing, however, have higher error. The error for the 64-entry buffer using selective indexing is higher than that using address mapping for more frequently executed blocks. This error, unlike that for the medium-executed blocks, is well distributed across all the blocks in this category and therefore does not affect any superblock formation decisions. As seen in Figure 3.8, the higher accuracy of the profile information assists in providing better performance for a 64-entry buffer using selective indexing (*Selec_64_1e6*) than that for address mapping (*Addr_64_1e6*). The increase in performance is more obvious for benchmarks for which the reduction in contentions is greater using selective indexing. The performance for *li* for a 64-entry profile buffer using selective indexing is about 25% higher than using address mapping. That for *sc* is about 14% higher. The performance for *compress* and *eqntott* employing superblock scheduling using the information obtained through the *selective indexing* approach are comparable to those obtained with *exact* profile information. However, for *gcc* the technique does not succeed in increasing performance by much due to the large number of inter-procedural contentions.

3.2.3 Compiler indexing

A further refinement of the profile buffer indexing scheme is possible if the profiling process can be provided with extensive compiler support. In the *selective indexing* approach, one bit in the branch instruction was used to indicate that a branch instruction should be profiled. However, due to the limitations of mapping into the buffer with the address of the branch instruction, the results obtained are in some cases inaccurate. To improve the situation, the *compiler indexing* method encodes the profile buffer index into the branch instruction as an added field in the instruction set architecture. This index indicates the entry in the *profile buffer* that is to be updated by the branch instruction. The compiler initially selects arcs for profiling as guided by the spanning tree algorithm for *selective indexing*. It then assigns indices to the branches to be profiled in such a way as to guarantee that adjacent or near-by branches do not map into the same entry. Compiler indexing can therefore further reduce the number of intra-procedural contentions.

The efficiency of the *compiler indexing* scheme is illustrated in Tables 3.5 and 3.6. The further reduction in intra-procedural contentions is indicative of the number of adjacent or near-by branches that caused the excessive contentions in the *address mapping* and *selective indexing* approaches. For example, the percentage of intra-procedural contentions for *gcc* is reduced to 7.29% (64-entries) for selective indexing to 4.35% . For *li* the contentions are reduced from 6.49% to 1.11% for a 64-entry profile buffer. Therefore the profile information obtained through this support from

Table 3.5: Percent contentions of original accesses after reduction in profiling branches and using *compiler indexing*. (Intra-procedural contentions)

Benchmark	Percent Contentions			
	8	16	32	64
compress	14.88%	0.00%	0.00%	0.00%
espresso	10.03%	3.46%	0.54%	0.12%
eqntott	0.15%	0.00%	0.00%	0.00%
gcc	21.39%	13.12%	8.17%	4.35%
li	20.23%	11.80%	5.83%	1.11%
sc	27.79%	9.54%	2.56%	0.38%
mean	15.75%	6.32%	2.85%	1.00%

Table 3.6: Percent contentions of original accesses after reduction in profiling branches and using *compiler indexing*. (Inter-procedural contentions)

Benchmark	Percent Contentions			
	8	16	32	64
compress	0.00%	0.00%	0.00%	0.00%
espresso	0.84%	0.69%	0.60%	0.46%
eqntott	0.00%	0.00%	0.00%	0.00%
gcc	19.35%	22.32%	20.11%	17.95%
li	16.36%	18.64%	19.12%	17.00%
sc	8.77%	7.09%	4.33%	3.58%
mean	7.55%	8.12%	7.36%	6.50%

the compiler is expected to be more accurate and consistent. However, a small increase is seen in the inter-procedural contentions.

The *average arc error* for the *compiler indexing* scheme as shown in Figure 3.11 is lower than that for the *selective indexing* technique. The technique yields lower errors than *selective indexing* for the medium and most-frequently executed basic blocks (i.e., execution weights 10^3–10^9) for a 64-entry buffer. Even a 32-entry profile buffer employing compiler indexing has less error for the highly executed blocks. Figure 3.10 shows the results of superblock scheduling using the profile information from *compiler indexing*. Performance using compiler indexing equals that possible using exact profile information for *compress* and *eqntott*. The speedup for *sc* increases by more than 10% for a 64-entry profile buffer over selective indexing. However, performance for *gcc* and *li* does not increase much due to the presence of a large number of inter-procedural contentions. The harmonic mean indicates that performance using a 64-entry profile buffer with compiler indexing is at least 7% better than that using selective indexing.

3.3 Associative Profile Buffer

The profile buffer due to its limited size suffers from contentions. The selective and compiler indexing techniques can reduce the contentions. However, even with the use of compiler indexing the number of inter–procedural contentions remains large, thus hurting performance. Associativity has been used in regular caches to reduce the miss ratio. In the same spirit, associativity in the profile buffer may be expected to

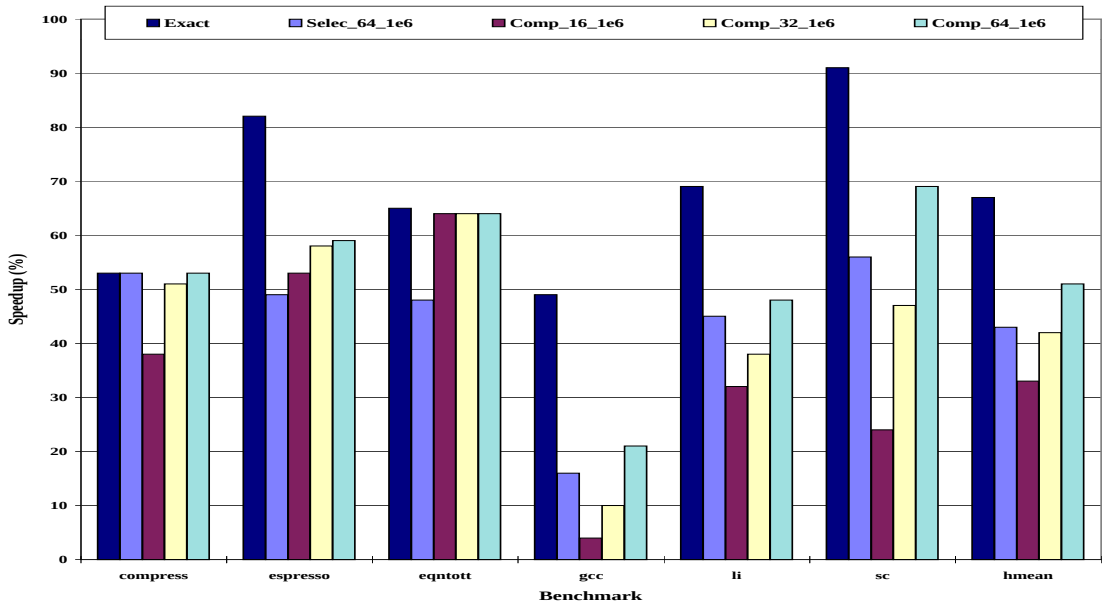

Figure 3.10: Speedup for superblock scheduling using profile information from the profile buffer with compiler indexing. (*Comp_n_i* indicates a profile buffer with n entries sampled at intervals of i instructions.)

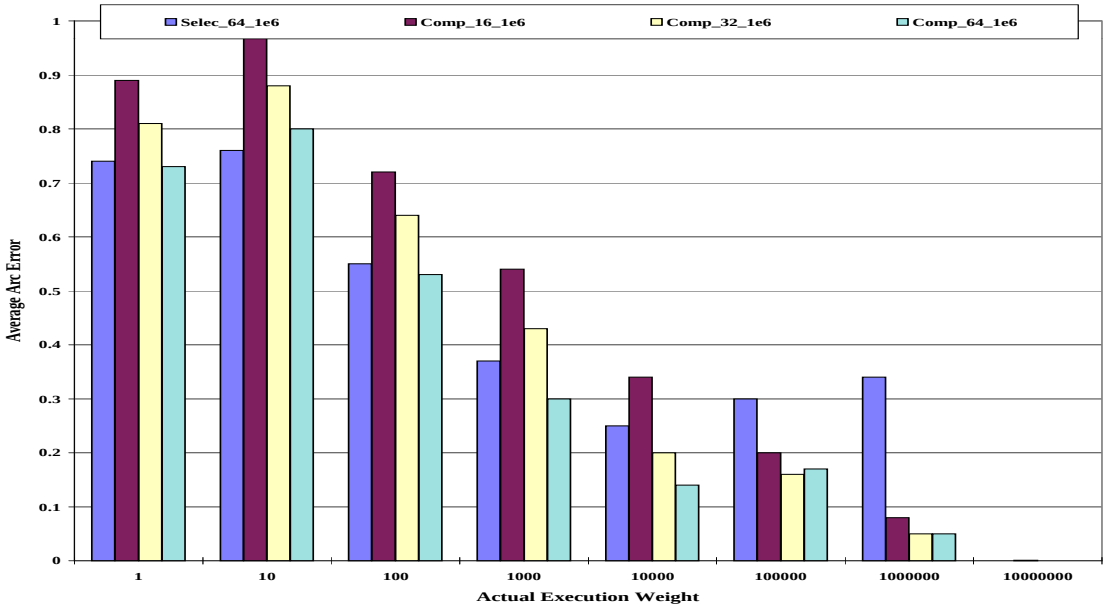

Figure 3.11: Average arc error in profile information from profile buffer with compiler indexing.

reduce the number of contentions and in doing so provide better performance. The following subsections investigate the effect of associativity in the profile buffer using the selective and compiler indexing techniques.

3.3.1 Selective indexing

The profile buffer experiments using selective indexing were repeated using a 4-way set-associative profile buffer. The performance results are shown in Figure 3.12. *Selec_4_assoc_64_1e6* indicates the results for a 4-way set-associative buffer with a total of 64 entries sampled at a sampling interval of 1 million instructions. The performance approaches that possible with perfect profile information for at least three benchmarks, namely, *compress*, *espresso* and *eqntott*. The performance increases by at least 25% for *espresso* and 15% for *eqntott* and *gcc* when compared to selective indexing without associativity (*Selec_64_1e6*). There is also a 10% increase for *sc*. The harmonic mean indicates that the addition of set associativity to the profile buffer using selective indexing provides at least a 10% performance increase.

3.3.2 Compiler indexing

Figure 3.13 shows the results for a 4-way set-associative profile buffer employing compiler indexing. The performance is comparable to that using exact profile information for *compress*, *espresso* and *eqntott*. For *gcc* even a 16-entry set-associative profile buffer provides more than a 10% improvement over a non-associative buffer. A

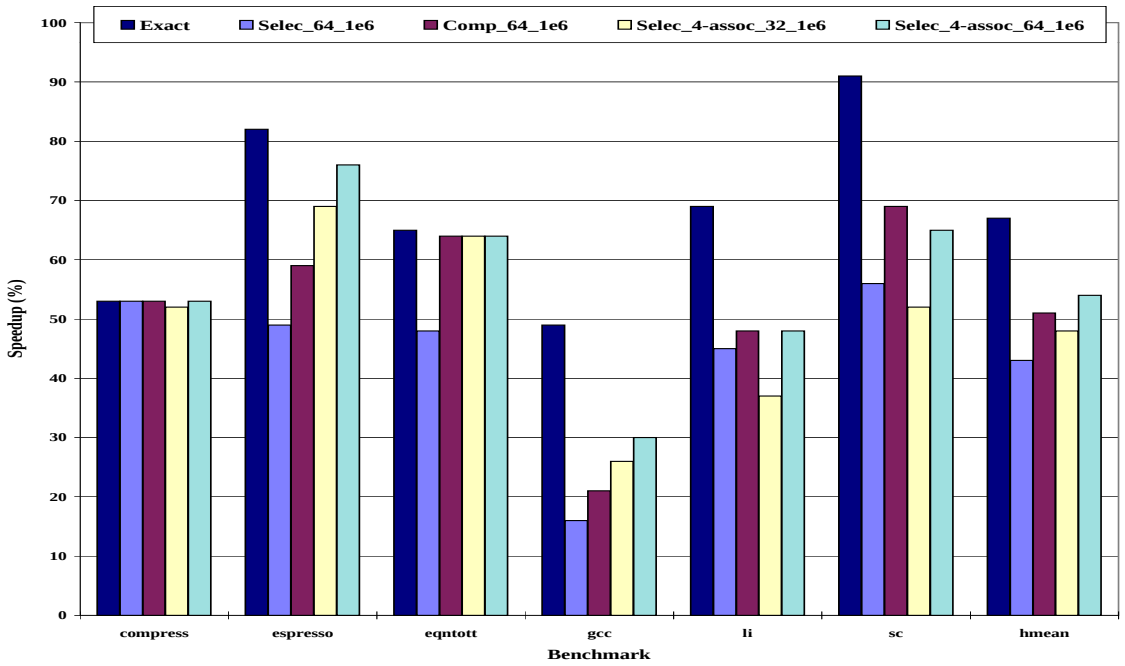

Figure 3.12: Speedup for superblock scheduling using profile information from the associative profile buffer with selective indexing. (*Selec_s-assoc_n_i* indicates an s-way set-associative profile buffer with a total of n entries sampled at intervals of i instructions.)

performance improvement of more than 20% is possible with a 64-entry set-associative buffer. A 10% improvement is also seen for *li*. By far the greatest performance improvement is for *sc*. At least a 30% improvement is seen for *sc* over a non-associative buffer using compiler indexing.

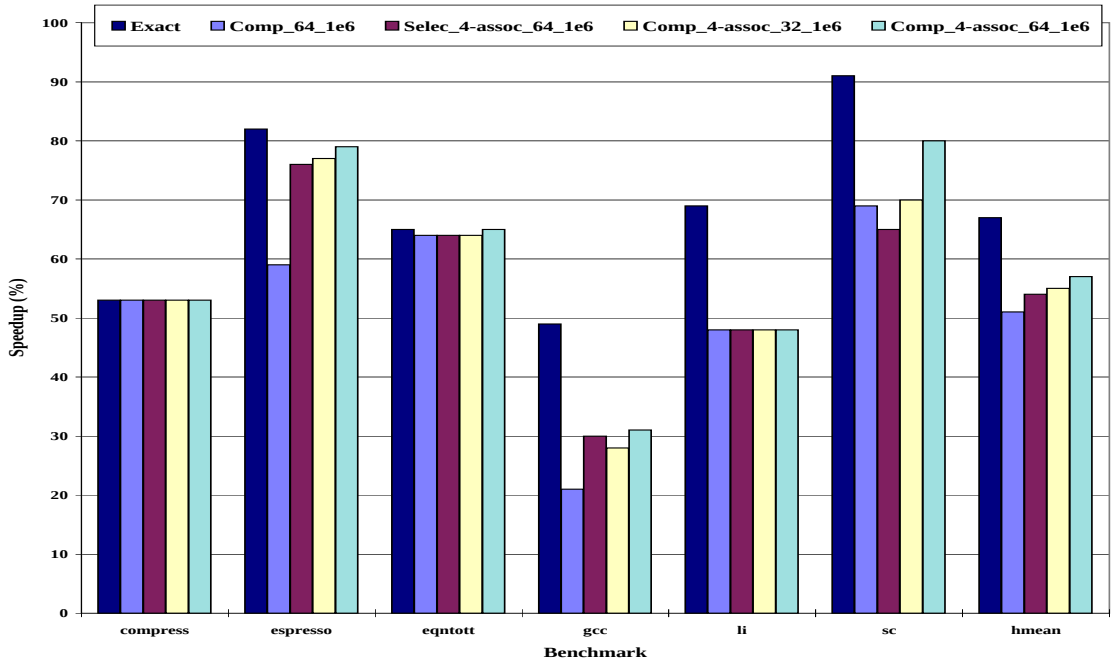

Figure 3.13: Speedup for superblock scheduling using profile information from the associative profile buffer with compiler indexing. (*Comp_s-assoc_n_i* indicates an s-way set-associative profile buffer with a total of n entries sampled at intervals of i instructions.)

3.4 Summary

Fast and non-intrusive methods are required for effective, efficient profiling in order to gain commercial acceptance of profiling. This chapter has further developed the ideas presented by Conte, *et al.* [12] for a hardware–based profiling technique that does not

significantly impact run–time. The goal of this paper was to improve accuracy of the information for compiler optimization while requiring minimal hardware. This was achieved by using a set of hardware based counters called the *profile buffer*. Three schemes for accessing the profile buffer were investigated. The first approach, *address mapping*, accessed the profile buffer for each and every dynamic branch, but suffered from excessive buffer contention for small buffers. The other two methods, *selective indexing*, and *compiler indexing*, limited the number of branches that could access the *profile buffer*. In the *selective indexing* scheme, one bit in the branch instruction was used to indicate a branch instruction to be profiled. In the *compiler indexing* scheme, however, a branch instruction contains an index into the *profile buffer*. This index is assigned at compile time.

The *address mapping* and *selective indexing* schemes are easily applicable to existing architectures without much overhead in cost. It is also possible to implement *selective indexing* without changes in the ISA if the ISA provides for complements of every branch condition (for example, branch equal (BEQ) and branch not equal (BNE)). In the *compiler indexing* scheme, a few additional bits are required depending on the size of the *profile buffer*. Recording the information in the buffer to memory on a context–switch is the only demand placed on the operating system. One other advantage of the profile buffer is that it is not in the critical path of the processor, allowing for a lazy-increment implementation of its counters to further reduce its implementation cost.

All of the proposed indexing approaches provide high performance when used for superblock scheduling, even though the size of the profile buffer required to generate moderately accurate and complete profile information is small. In fact, even small profile buffers of 16-32 entries employing *selective indexing* or *compiler indexing* with associativity perform well. When coupled with alpha/beta testing software development processes that are already used by most commercial software vendors, the low impact of hardware–based profiling using the profile buffer becomes a viable technique to aid profile-driven optimization.

Chapter 4

Issues in Trace Formation

Previous chapters have detailed the use of branch prediction hardware or custom hardware for the purpose of profiling by sampling the hardware on context-switches or timer interrupts. Some measure of the validity of the profile information obtained by such methods is required. It needs to be confirmed that profile information obtained using such hardware-based profiling methods is indeed representative of the actual profile for the program being profiled. This chapter addresses these concerns. Since the hardware-based profiling process is similar in nature to statistical sampling, this chapter uses statistical sampling theory to justify the claim that representative profiles may be obtained by sampling the contents of hardware structures in the processor.

An optimizing compiler requires to detect in some manner the precision of the profile information for a given branch in the program. The optimization using the profile information can then make a decision based on this precision measure whether

the information provided to it is representative of the true profile. Confidence interval analysis may be used to measure the precision of the profile information, leading to smarter trace formation decisions. This chapter discusses confidence interval analysis and the manner in which it may be used in an optimizing compiler using profile data obtained by sampling profiling hardware.

Trace formation algorithms depend on relative frequencies of the nodes and arcs in a control-flow graph. Profiling using branch prediction or custom hardware such as the profile buffer attempt to obtain accurate absolute counts. The indexing methods for such hardware do not consider the accuracy of the weight of a basic block in relation to its neighboring basic blocks. This chapter describes another hardware-based profiling method which operates by using hardware to profile execution frequencies while trying to preserve the integrity of the execution frequencies of nodes and edges adjacent to the profiled node or arc. The structure of the hardware is a first in first out queue and is therefore named the *history FIFO*.

4.1 Statistical Sampling

Sampling has been defined as the process of drawing inferences about the whole by examining a part [44]. It is a technique frequently used by statisticians in estimating characteristics of large populations to economize on time and resources. Sampling may be broadly classified into two types, *probability sampling* and *non-probability sampling*. Probability samples contrast with non-probability samples in that they are

chosen by a randomized mechanism. This assures selection of samples independent of subjective judgments. *Simple random sampling* is known to be one of the most accurate methods for sampling large populations. It involves a random selection of single elements from the population. However, the cost associated with this technique makes its application infeasible in some cases. Another less accurate, but cost-effective technique is *cluster sampling*. This technique collects contiguous groups of elements at random intervals from the population.

An element on which information is required is known as a *sampling unit*. For the purpose of this thesis, a sampling unit is a single execution of a branch. The variable being measured is the outcome of the branch. The various instances of a branch execution may be thought of as Bernoulli variables, since they have only two possible values, *taken* (1) and *not-taken* (0), with probability p of being *taken*. A parameter is a numerical property of the population under test. The parameter for this study is the *proportion* (p) of the number of times a branch was *taken* to the number of times the branch was executed. The sample estimator for the population proportion p is the sample proportion $\hat{p}$, which is the ratio of the number of branch executions in a sample that are *taken* to the total number of executions of the branch captured in the sample. Therefore,

$$\hat{p} = \frac{Num\_taken_{sample}}{Total\_executions_{sample}} \tag{4.1}$$

The total number of a given branches execution in the sample is the sum of the

number of times the branch is *taken* and *not-taken* in the sample. The sampling distribution is the probability distribution of the sample estimator, $\hat{p}$.

Consider the execution of a single branch in the processor. Each execution of the branch is a sampling unit. The total number of executions of the branch constitute a complete list of the sampling units or what may be termed as the *total population* for that branch. Simple random sampling involves random selection of sampling units from this list for inclusion in the sample. The interval between two sampling units included in the sample is randomized. The sampling unit immediately following each interval is included in the sample. To be able to extract single executions of a branch at random intervals can be expensive. One example of such sampling is *program counter sampling* which records the program counter at each sampling point. Each sampling point implies overhead in time. Program counter sampling can potentially incur a high overhead when collecting a sizeable sample. An alternative method is to extract subsets of the branch execution. The results of the execution of a branch are collected in the processor and then recorded to memory at a sampling point. This method records results of multiple executions of a branch to memory at a sampling point. This method of sampling is essentially *cluster sampling*. The sampling units, branch executions, are extracted in clusters. The profiling techniques described in previous chapters using branch prediction or custom hardware such as the profile buffer are essentially applications of cluster sampling. For example, when profiling using two-level prediction hardware with a branch history register size of n bits, n

actual executions of a branch are captured. This implies that the executions of the branch are captured in clusters to be included in the sample with a cluster size of n.

4.2 Sample design

Since clusters from different locations may be selected from sample to sample, the estimates may vary across repeated samples (i.e., across repeated sampled simulations). Repeated samples yield values of proportions that form a distribution. This distribution is known as the *sampling distribution*. Statistical theory states that, for a well designed sample, the mean of the sampling distribution is representative of the true proportion. Sampling bias is measured as the difference between the mean of the sampling distribution and the sample proportion. It is a result of the sampling technique employed and the sample design. Sample design involves the choice of a robust (*i*) *sample size*, (*ii*) *cluster size* and, (*iii*) *number of clusters*. Sampling techniques and the estimates derived from them may be prone to excessive error if the sample is not properly designed. Increasing *sample size* typically reduces sampling bias. In case of cluster sampling, *sample size* is the product of the *number of clusters* and *cluster size*. Of these two, the *number of clusters* should be increased to reduce sampling bias, since it constitutes the randomness in the sample design.

The limited size of the hardware in the hardware-based profiling schemes presented in previous chapters lead to contentions. The contentions result in some branches being denied adequate representation in the hardware, and therefore being sampled

a fewer number of times. This implies that the sample for such branches which are victims of contentions contain a fewer number of clusters, thereby introducing bias into the sample collected. In the 2-level profiling scheme, the cluster size is fixed due to the fixed size of the *branch history registers*. The profile buffer being smaller in size than branch prediction hardware suffers more contentions leading to a reduction in the number of clusters. It attempts to compensate for the loss in the number of clusters collected by using counters and thus increasing the size of the clusters collected. One solution to overcoming the problem of contentions and consequently the reduction in the number of clusters collected for each branch is to increase the sampling interrupt rate.

An additional consequence of the selection of clusters at random is *sampling variability*. The standard deviation of the sampling distribution is a measure of the variation in estimates that might be expected across samples. Sampling bias and variability can be reduced by making the clusters internally heterogeneous, (i.e., large standard deviation of the parameter within the cluster), making the cluster proportions homogeneous, and by increasing the number of clusters [45],[44]. The reduction of bias requires that the design of the sample be robust and all factors that could increase error be taken into consideration.

4.3 Confidence Intervals

Assume n executions of a branch and assume $X_1, X_2, \ldots X_n$ to be the corresponding values of the measured variable for each execution of the branch. Since the branch and therefore, the measured variable can only have one of two outcomes, *taken* (1) and *not-taken* (0), the variable Num_taken_{sample} in Equation 4.1 is a sum of random variables. Therefore, regardless of the form of the underlying distribution of observations, as sample size increases, the distribution of the sample proportion approaches a normal distribution [44]. This property makes it possible to use the normal distribution to approximate the sampling distribution of the proportion when sample sizes are "large". The Central Limit Theorem provides a normal approximation to the distribution of Num_taken_{sample}, and therefore to the distribution of $\hat{p}$. The mean of the approximation is given by,

$$\mu_{\hat{p}} = p \tag{4.2}$$

Bias exists in every sample due to the random nature of the sample. It is possible to predict the extent of the error caused by this bias. The *standard error* of the parameter under consideration is used to measure the precision of the sample results (i.e., the error bounds) [44]. Standard error is a measure of the expected variation between repeated sampled runs using a particular sampling regimen. These repeated simulations yield proportions that form a distribution. The standard error

is defined as the standard deviation of this distribution. The properties of the normal distribution can be used to derive the error bounds for the estimate obtained from a simulation. The standard error for a population proportion p is given by,

$$\sigma_{\hat{p}} = \sqrt{\frac{p(1-p)}{n}} \tag{4.3}$$

When only a sample is available, the proportion (p) for the total population, is unknown. Therefore the standard error $\sigma_{\hat{p}}$ is unknown as well. In such situations p in Equation 4.3 may be substituted with $\hat{p}$, the estimated sample proportion. This is termed as the *estimated standard error* and is denoted by $SE_{\hat{p}}$. The estimated standard error is computed as

$$SE_{\hat{p}} = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{4.4}$$

The sampling distribution may be assumed to have a normal distribution when the population being considered, i.e. the number of executions of a branch, is *large*. When a population is too small, as may be the case with the execution instances of some branches, the sample size may constitute a large part of the population. Therefore, the standard error needs to be corrected. The finite population correction factor is applied to the estimated standard error as the correction. The estimated

standard error becomes,

$$SE_{\hat{p}} = \sqrt{\frac{N-n}{N-1}}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{4.5}$$

A confidence interval is a device for pinpointing the *true value* of a parameter within a range of values determined from a random sample. In other words, it is a range of values of a population parameter that will be assumed to contain the true parameter value. A probabilistic *figure of merit*, called a *confidence coefficient* is chosen in advance of the experiment. This coefficient specifies the probability that the confidence interval will cover or contain the true parameter value. A confidence interval is characterized by a lower limit L and an upper limit U.

A 95% confidence interval may be interpreted as follows: If the experiment of taking a random sample (of given size) from the population and calculating the values of L and U is repeated a large number of times, the interval will contain the true parameter value ($L \leq p \leq U$) for about 95% of the samples. The limits of a confidence interval for a proportion p are given by

$$L = \hat{p} - zSE_{\hat{p}} \tag{4.6}$$

and

$$U = \hat{p} + zSE_{\hat{p}} \tag{4.7}$$

Therefore, substituting for $SE_{\hat{p}}$ in Equations 4.6 and 4.7,

$$L = \hat{p} - z\sqrt{\frac{N-n}{N-1}}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{4.8}$$

$$U = \hat{p} + z\sqrt{\frac{N-n}{N-1}}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{4.9}$$

where z is the critical value for the chosen confidence coefficient, and is equal to 1.96 for a 95% confidence interval.

When the true value of p is near 0 or 1, the confidence interval computations in previous equations are somewhat inaccurate. The true probability with which the given limits contain p is smaller than the quoted confidence coefficient. More accurate limits are provided by Blyth and Still [46] as,

$$L = \frac{(x-.5) + z^2/2 - z\sqrt{(x-.5) - (x-.5)^2/n + z^2/4}}{n+z^2} \tag{4.10}$$

$$U = \frac{(x+.5) + z^2/2 + z\sqrt{(x+.5) - (x+.5)^2/n + z^2/4}}{n+z^2} \tag{4.11}$$

where x is the number of successes and n the number of trials.

Figure 4.1 shows the percentage of branches for which the true proportion lies

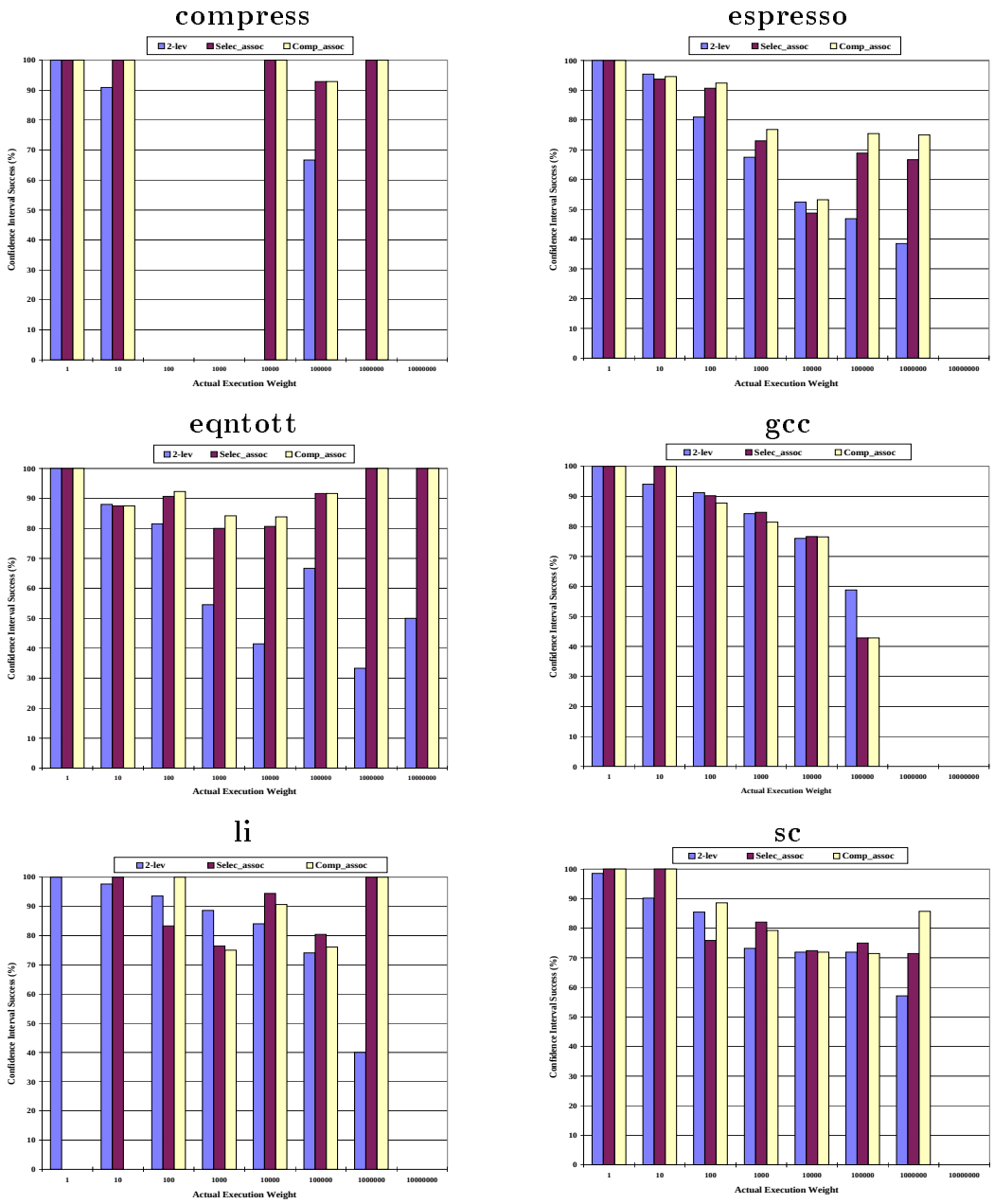

Figure 4.1: Percentage of branches for which the true proportion lies in the 95% confidence interval. The percentage is plotted for different execution frequencies.

within the 95% confidence interval. The percentage is plotted for branches at different execution frequencies. An execution frequency of 100 implies a branch that is executed between 100-999 times. The results for different hardware-based profiling techniques are plotted, namely, profiling using two-level branch prediction hardware (*2-lev*), 64-entry 4-way set-associative (16×4) profile buffer using selective indexing (*Selec_assoc*) and a 64-entry 4-way set-associative (16×4) profile buffer employing compiler indexing (*Comp_assoc*).

4.3.1 Trace formation using confidence intervals

Optimizations such as superblock scheduling that use profile information require to verify that the information provided by sampling hardware is representative of the actual profile. Smarter optimization decisions can then be made based on the information. The length of a confidence interval is a measure of the precision with which the confidence interval may be expected to contain the true parameter value. The length l is given by,

$$l = U - L \tag{4.12}$$

The compiler can compute the confidence interval for each branch in the program and decide whether to utilize the profile information for that branch based on the profile information. Profile information for a branch which yields a narrow confidence interval is to be preferred over that which yields a wider interval. A threshold may be

chosen for the length of the confidence interval. Given the profile information for a branch the information may be used if it yields a length of confidence interval below this threshold and discarded if it does not. Exploration of this topic is beyond the scope of this thesis.

4.4 The History FIFO

Trace formation algorithms generally depend on the relative frequencies of basic blocks and the edges between them. The frequency of an edge is compared with the frequencies of the other edges out of a basic block to decide which edge and target node to include in the trace. The frequency of a source node is also compared with the frequencies of adjacent blocks in order to determine if trace formation should be continued. If the relative frequencies of adjacent blocks are small, the trace formation process is stopped. This is done to avoid code bloat. An example of such an algorithm is *superblock formation*. Therefore, for most trace formation algorithms, relative frequencies rather than absolute frequencies are important.

The methods elaborated on in previous chapters for profiling branches do not attempt to preserve the integrity of relative frequencies. The profile buffer as well as prediction hardware are accessed using indices that are computed in isolation from those of the blocks or arcs adjacent to it. There arise cases in such profiling methods where it is possible that the sum of the weights on the incoming arcs of a block is not equal to the sum of the weights on the outgoing arcs of the block. Such instances can

mislead the trace formation algorithm. The *history FIFO* presented in this section is a queue structure which attempts to maintain accurate relative frequencies. The design of the *history FIFO* is presented in the next subsection. Subsection 4.4.2 explains an alternate implementation of the *history FIFO*. Performance results are presented in subsection 4.4.3.

4.4.1 Design of the history FIFO

The *history FIFO* maintains a queue of branch addresses in the processor. This queue is modeled as a first-in first-out queue. Another structure used is the *branch result queue*, also modeled as a first-in first-out queue. This queue contains the actual outcomes of the branches in the program. When a branch is resolved the address of the branch in inserted into the *address queue* and the outcome of the branch (*taken* or *not-taken*), is inserted into the *branch result queue*. Figure 4.2 shows a control flow graph that is profiled by the hardware. The path traversed through the control flow graph goes through nodes $N_1, N_2, N_4, N_5, N6, \ldots$. The actual outcomes for each of the branches in the path is recorded in the *branch result queue* and the addresses of the branches are recorded in the *address queue*.

The hardware is sampled either on a context-switch or a timer interrupt. At every sampling point the address at the head of the *address queue* and the complete contents of the *branch result queue* are read out to memory. This information can then be used to assign weights to the control flow graph. In Figure 4.2 the address at

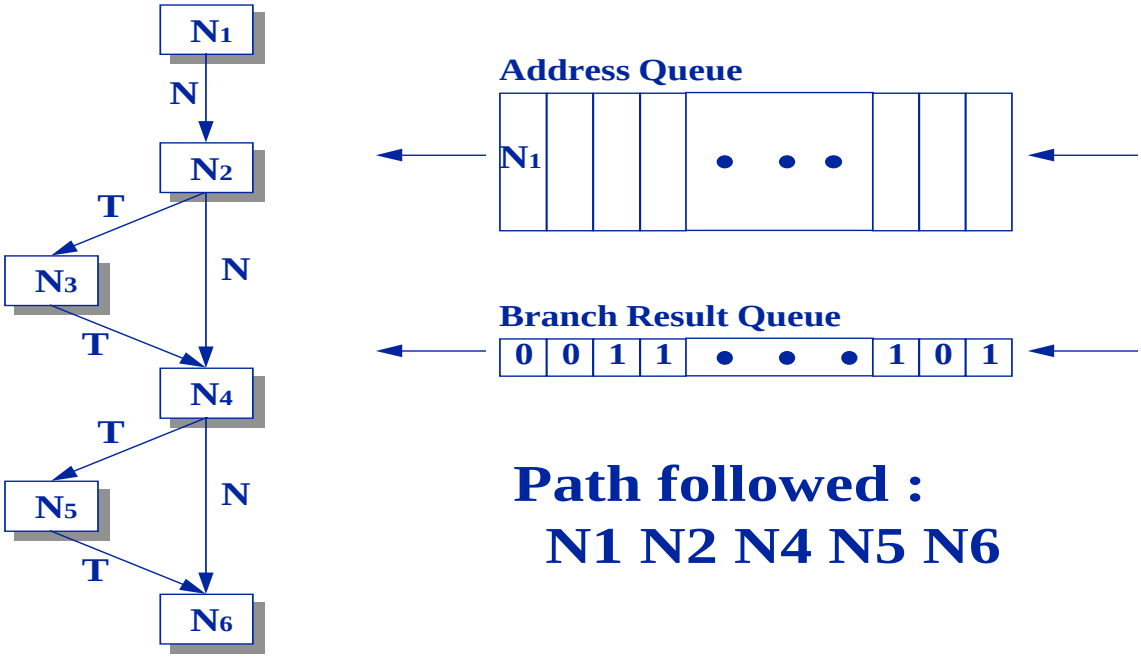

Figure 4.2: The design of the history FIFO.

the head of the *address queue* is the address of the branch in N_1. Using this address, the pattern in the *branch result queue* and the knowledge of the control graph, the path traversed through the graph can be determined.

4.4.2 Alternate implementations

The *address queue* in Figure 4.2 is used to determine the start of the partial control flow graph for which the *branch result queue* holds the branch result pattern. The *address queue* may imply considerable hardware cost. This cost may be alleviated by eliminating the *address queue*. In the absence of the *address queue*, the program counter and the contents of the *branch result queue* are recorded at each sampling

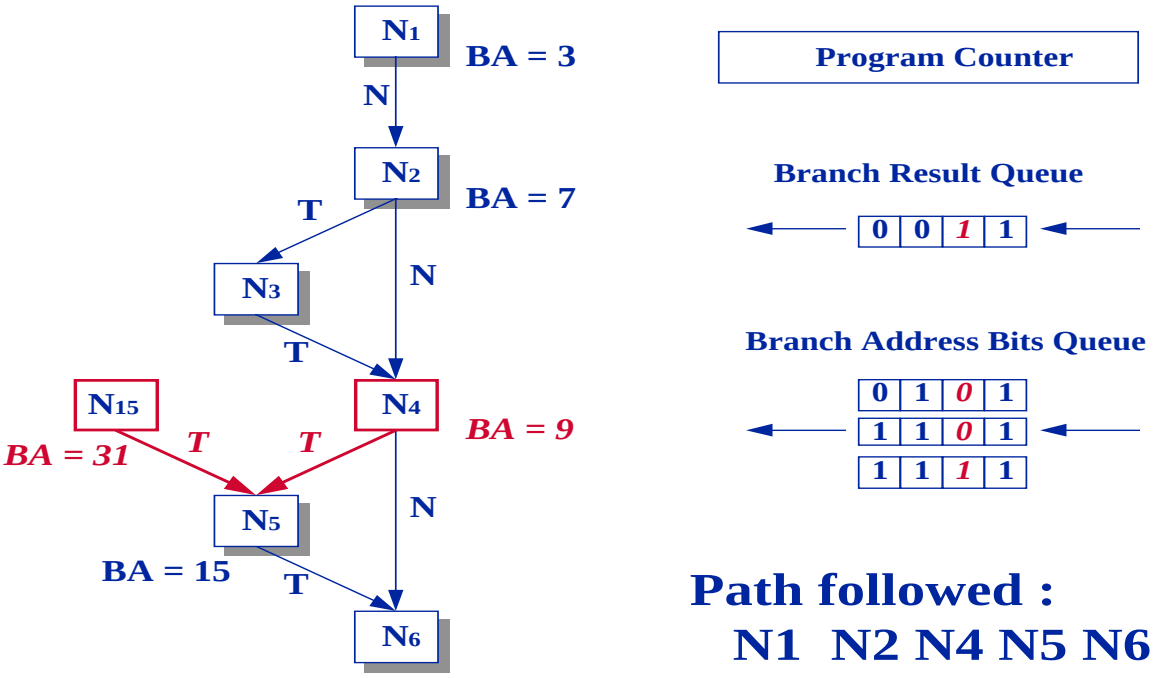

Figure 4.3: Alternate implementation of the history FIFO.

point. The profile information for a control flow graph can then be reconstructed by determining the block that contains the program counter and using the results of the *branch result queue* in a direction opposite to that in which the arcs in the graph were traversed.

This technique suffers when a block is the *taken* target of two different blocks. If more than one incoming arc in the block is a *taken* arc it may not be possible to determine the arc that was traversed using the result in the *branch result queue*. When the branch in any of the multiple source blocks with the same *taken* target block execute, the result recorded in the *branch result queue* will be *taken*. Hints are required to determine which of the incoming arcs was actually traversed during

Table 4.1: Percentage of instances where a block has multiple incoming *taken* arcs where at least n bits, where $n = 1, 2, \ldots, 8$ are required to disambiguate the sources.

Benchmark	Number of bits required							
	1	2	3	4	5	6	7	8
compress	21.3	2.4	4.0	1.9	0.5	0.8	0.3	0.0
espresso	33.0	3.0	1.9	0.9	0.6	0.2	0.1	0.1
eqntott	22.3	3.6	1.7	1.4	0.6	0.6	0.4	0.0
gcc	18.0	3.1	2.9	1.5	0.8	0.5	0.3	0.2
li	19.7	4.2	1.9	0.9	0.7	0.5	0.1	0.0
sc	20.9	2.8	2.0	1.2	0.8	0.3	0.1	0.1
mean	22.5	3.2	2.4	1.3	0.7	0.5	0.2	0.07

execution. Figure 4.3 shows the history FIFO as a combination of the *program counter* and the *branch result queue*. The figure also shows a *branch address bits queue*. This queue is similar to the *address queue* in Figure 4.2. However, each entry in the queue contains only a few bits from the address of each branch that was executed, rather than the complete address as in the *address queue*. These bits can be used as hints to disambiguate between two incoming *taken arcs*. Note that the cost of the *branch address bits queue* is far less than the address queue since it stores only a few bits rather than the complete address. The complete address in a 64-bit architecture could mean up to 64 bits. The number of bits to be stored in each entry of the *branch address bits queue* can be determined empirically using analysis of code that is representative of the workloads the processor may be expected to execute in practice. Table 4.1 shows the results of such analysis for the six benchmarks used here. The percentage of instances where $1, 2, \ldots, 8$ bits are required to distinguish between multiple source blocks are shown. The results are obtained through static analysis. Most often a

single bit is adequate (22.5%). Two to three bits suffice to handle most of the cases in these benchmarks.

4.4.3 Performance

The performance for the *history FIFO* is compared against *program counter* sampling since the techniques are similar. Program counter sampling reads the program counter to memory at regular intervals. The *history FIFO* on the other hand reads out multiple stored branch addresses. Profiles obtained using program counter sampling are coarse-grain in nature. They may be used to aid optimizations such as code-layout, which do not need fine-grain profile information. Trace formation and scheduling optimizations require finer-grain information such as the relative frequencies of branches. The history FIFO attempts to provide this kind of information.

Figure 4.4 indicates that information obtained from *program counter* sampling at a fast interrupt rate of 100,000 instructions (*PC_1e5*) is not adequate to satisfy the requirements of a trace formation and scheduling optimization. Using the profile information using program counter sampling degrades the performance for *gcc* and *sc*. A 64-entry *history FIFO* at an interrupt rate of 100,000 instructions (*HF_addr_1e5*) as well as at an interrupt rate of 1000,000 instructions (*HF_addr_1e6*) outperforms program counter sampling across all benchmarks. However, the performance for *HF_addr_1e6* reduces to about 50% of *HF_addr_1e5*. Figure 4.4 also shows speedups obtained using selective indexing as explained in Chapter 3 at interrupt rates

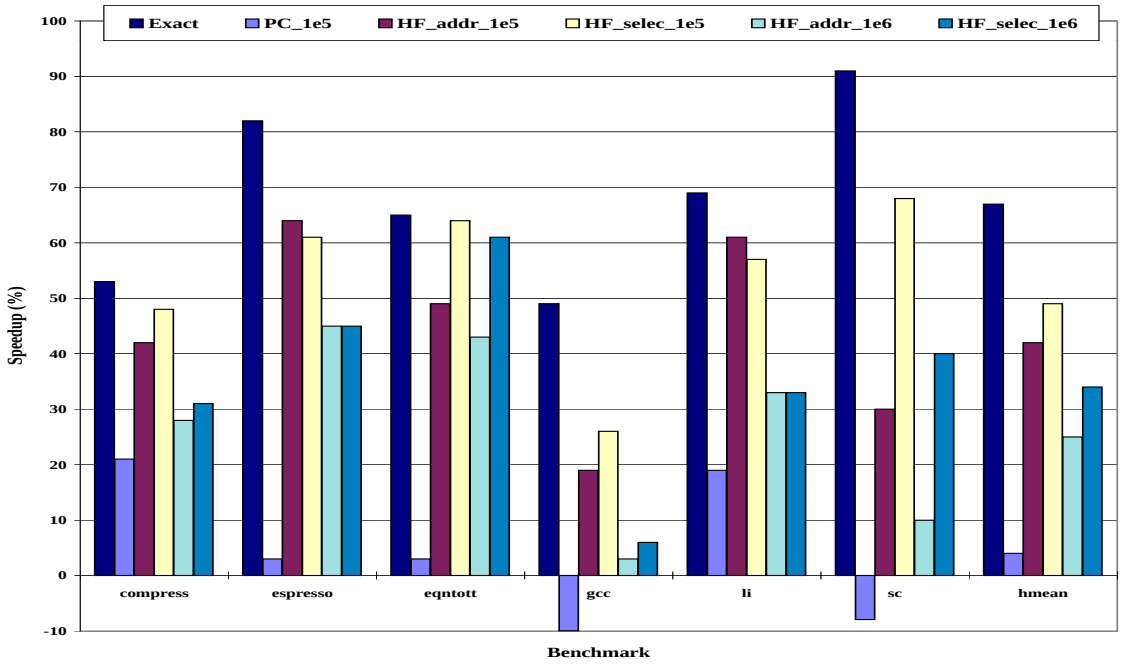

Figure 4.4: Performance using a 64-entry *history FIFO* at interrupt rates of 100,000 and 1000,000 instructions compared against performance obtained using *program counter sampling* at an interrupt rate of 100,000 instructions.

of 100,000 instructions (*HF_selec_1e5*) and 1000,000 instructions (*HF_selec_1e6*). Performance is greatly enhanced by using selective indexing for *eqntott* and *sc*. Overall, a performance improvement is observed using selective indexing.

4.5 Summary

This chapter uses statistical analysis to measure the validity of the profile information obtained from hardware-based profiling methods. The application of statistical theory to the analysis of profile information is detailed. Confidence intervals are suggested as a tool to measure the precision of the profile information obtained from hardware-based profiling methods. This chapter also explains the importance of keeping the relative frequencies of adjacent basic blocks and arcs in a control-flow graph consistent. A hardware-based profiling technique that attempts to preserve the consistency of weights (the *history FIFO*) has been described. Performance results for superblock scheduling using information from the history FIFO are presented and compared to superblock scheduling using profile information from program counter sampling.

Chapter 5

Memory Dependence Profiling

Speculative execution refers to the execution of an instruction before it is known that the result of the instruction is required. Speculative execution of instructions may be implemented as a dynamic mechanism controlled by the processor. Alternatively, it can be engineered statically during compile time. Compiler controlled speculation is gaining in importance since it can be used even in the presence of dynamic speculation mechanisms in the processor. The extent to which an instruction can be speculated dynamically is restricted by hardware limitations. Compiler controlled speculation however, can operate within a larger scope and therefore yield a greater performance benefit than dynamic speculation alone.

Speculative execution can be classified into control speculation and data speculation [4], [47]. Control speculation is achieved using some form of control prediction. Modern day processors implement dynamic control speculation by predicting the di-

rection of a branch in the control flow. The instructions on the predicted path are fetched into the processor and executed. The prediction for a branch is checked when the branch is resolved. If the branch was mispredicted, the results of the speculated instructions is nullified. Compiler-controlled control speculation is achieved by prediction branches using static profiles of the program or by using profiles obtained from a previous run. Fast, hardware-based methods for collecting profile information for a program were addressed in the preceding chapters of this thesis. This chapter addresses the subject of profiling for compiler-controlled data speculation.

Compiler controlled speculation has to obey the dependences between instructions. Memory references are one source of dependences. Such dependences are termed *memory dependences*. Memory dependences are a result of memory operations accessing the same locations in memory. The locations accessed are in many cases known only during run-time. However, analysis techniques can derive many such dependences. Such analysis is popularly termed *memory disambiguation*. Perfect memory disambiguation would result in the knowledge of all dependences through memory. However, perfect memory disambiguation is not possible in most cases because of the complexity involved and time restrictions. In the absence of perfect memory disambiguation, any two memory operations which cannot be conclusively disambiguated require to be considered as accessing the same memory location. Such memory operations, may not always access the same memory location resulting in unexploited performance.

Data speculation refers to the speculative execution of memory operations. A memory operation for which it cannot be determined whether the location accessed by it is the same as its preceding operations, may be speculatively executed before the preceding operations. However, when a dependence is violated execution needs to be altered to repair the mis-speculation. Repairing mis-speculation involves re-executing the speculated memory operation and all the operations dependent on it. The repair process penalizes the execution time. A pair of memory operations which conflict more often than not may result in performance degradation rather than improvement. Information from prior runs of the program can be fed to the compiler to enable it to make smarter data speculation decisions. This chapter deals with the collection of such data dependence profile information.

5.1 Memory profile buffer design

Data speculation is most profitable when applied to *potential* flow-dependences in memory operations. Instructions involved in a computation generally depend on a memory read operation (*load*). Speculation of a *load* instruction enables the speculation of other instructions dependent on it. The memory dependence profile buffer targets such flow dependences and obtains information on the frequencies with which a *load* instruction accesses the same location (*aliases*) with a preceding memory write (*store*) operation.

The structure of a basic memory profile buffer is shown in Figure 5.1. The buffer

is indexed using the address of the memory location *data address* accessed by the memory operation. The buffer may be direct-mapped or associative. It consists of a tag store. An entry corresponding to an entry in the tag store contains the address of the most recently encountered *store* operation which accessed data at the location in the tag store entry. It also contains the addresses of one or more *load* instructions which access the same location in memory. (Only one *load* address entry is shown in Figure 5.1). Counters are associated with each of the load or store entries. These counters are incremented when the corresponding *load* or *store* is encountered by the memory profile buffer. The actions associated with a *store* and *load* operation are as enumerated below.

The actions taken for a store instruction are as follows:

- The disambiguation profile buffer is accessed using the data address.
- The data address is loaded into the *data address* field of the selected entry.
- The address of the store instruction is loaded into the *store address* field of the selected entry.
- The *valid* bit for the entry is reset.

The actions taken when a load instruction is encountered are as follows:

- The buffer is accessed using the address of the data being loaded.
- If the data addresses in the *data address* field of the selected entry and that for

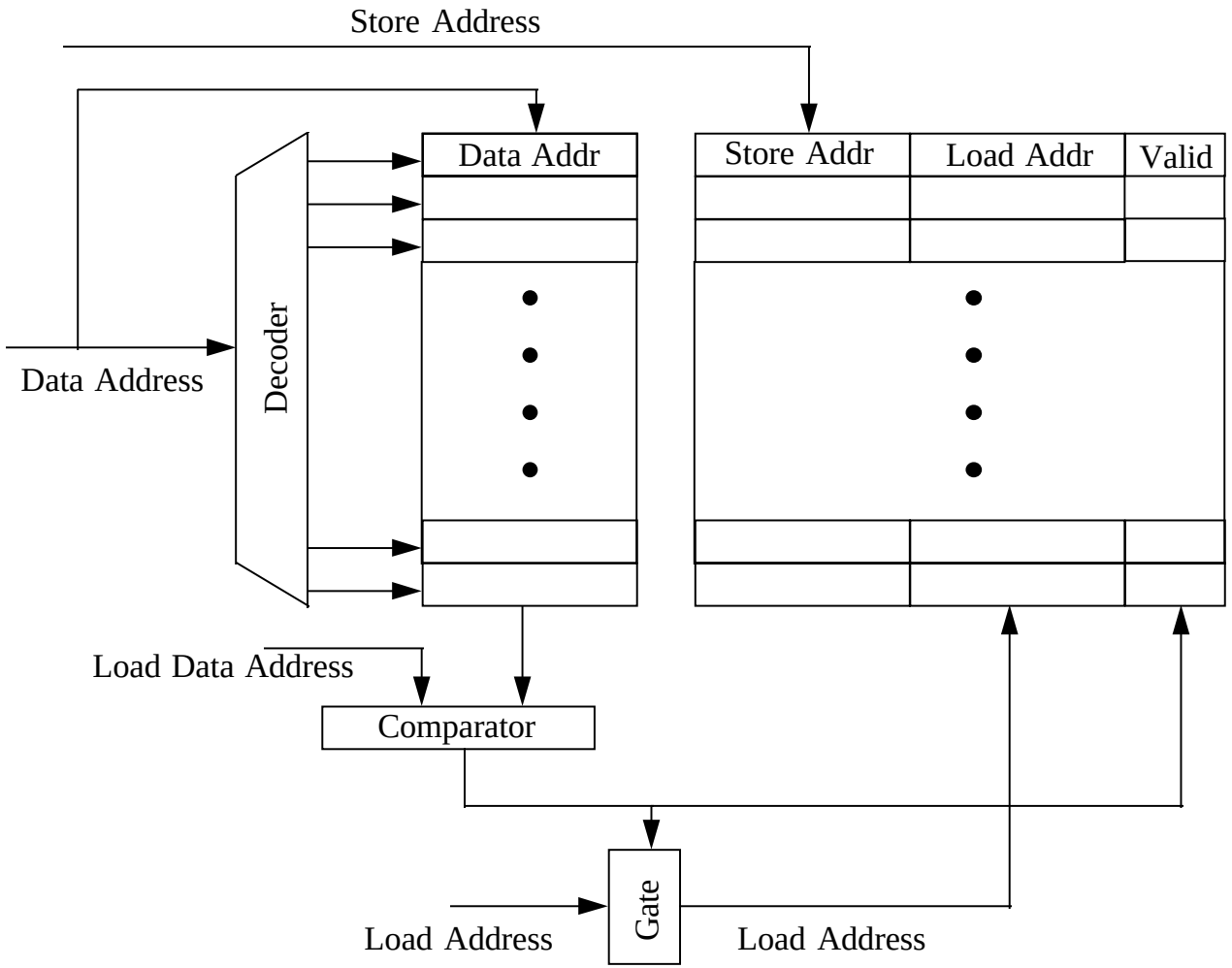

Figure 5.1: Hardware configuration for data reference profiling for memory disambiguation.

the load match, and the valid bit is not set, the address of the load is recorded in the *load address* field.

- The *valid* bit for the entry is set to indicate that an ambiguous store and load were found for the data address for the entry.

On a timer interrupt, the contents of the *store address* and *load address* fields of entries whose valid bit is set are read out to memory. This operation is performed by the routine which services the timer interrupt.

5.2 Alternate implementation

The amount of information recorded in memory may be reduced by using the configuration shown in Figure 5.2. It shows a single entry in the memory profile buffer. The space required to store the addresses of *load* instructions can be reduced by storing only the offset of the *load* from the *store* on which it is dependent. When the information in the memory profile buffer is used the actual address of the *load* instruction can be computed by adding the stored offset to the address of the *store* instruction. The distance of a recorded *load* instruction from the *store* instruction could be limited by the hardware. Therefore, it is necessary that a offset length be chosen such that most of the *load* instructions which are potential candidates for speculation be captured. The detailed actions taken when a memory operation is encountered by such a buffer are enumerated below.

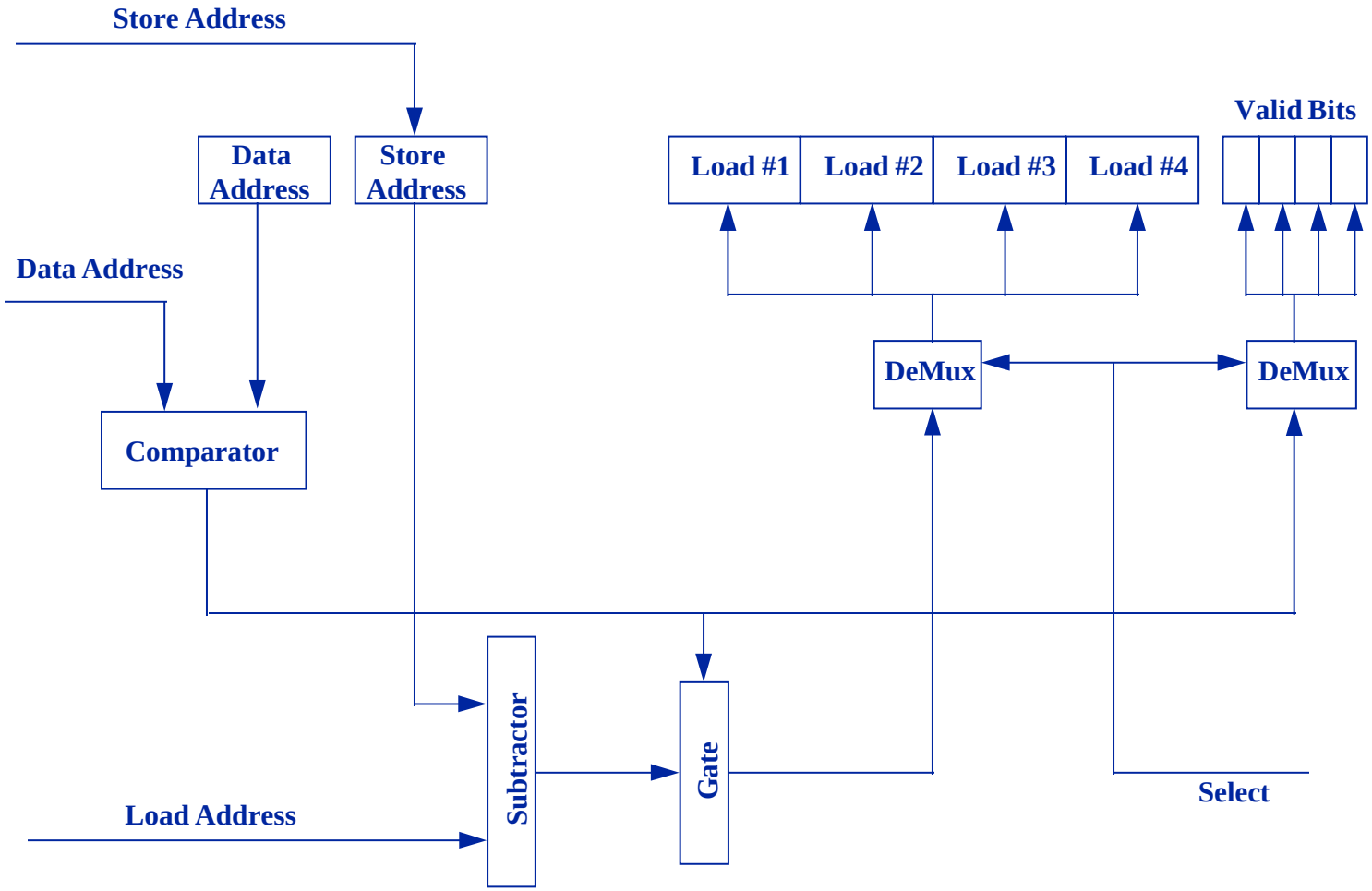

Figure 5.2: Alternate hardware configuration for data reference profiling for memory disambiguation.

The actions taken for a store instruction are as follows:

- The disambiguation profile buffer is accessed using the data address.
- The data address is loaded into the *data address* field of the selected entry.
- The address of the store instruction is loaded into the *store address* field of the selected entry.
- All the *valid* bits for the entry are reset.
- The count in the *distance counter* field is reset.

On every instruction that follows, the count in the *distance counter* field is incremented. The actions taken when a load instruction is encountered are as follows:

- The buffer is accessed using the address of the data being loaded.
- If the data addresses in the *data address* field of the selected entry and that for the load match, the count in the *distance counter* is recorded in one of the *load #* fields for which the corresponding valid bit is not set.
- The *valid* bit corresponding to the *load #* filed that is loaded is set.

On a timer interrupt, the contents of the *store address* and *load #* fields whose valid bit is set are read out to memory. This operation is performed by the routine which services the timer interrupt.

5.2.1 Selective Memory Profiling

Collecting profile information for every *store-load* pair during a programs execution can over-burden the memory profile buffer. Programs can contain *store-load* pairs that are easily disambiguated. Stack variables are allocated locations by the compiler and therefore may be statically disambiguated. Spill code generated during register allocation need not be profiled by the memory profile buffer since such *store* and *load* instructions are generated by the compiler and are easily disambiguated by the compiler. Excluding such *store* and *load* instructions from accessing the memory profile buffer can reduce the number of contentions in the buffer and consequently result in better and more accurate profile information.

In order to be able to indicate to the hardware the *store* and *load* instructions that should access the memory profile buffer a bit is required in the *instruction set architecture* encoding for memory operations. Only memory operations with this bit set would access the memory profile buffer. The results presented below are obtained using selective memory profiling assuming the presence of the additional bit in the encoding for memory operations.

5.3 Experimental Results

The accuracy of the memory profile buffer was measured for the benchmark 099.go from the SPEC95 integer suite of benchmarks. Table 5.1 shows the results. The

column *conflict frequency* indicates the percentage of time a *load-store* pair conflict (access the same location in memory). The column *buffer* indicates the configuration of the buffer used. Four buffers with 64 entries but varying associativity were experimented with. For example, a 32×2 buffer implies a 32-entry, 2-way set-associative buffer. For the buffers with associativity, the replacement algorithm used was *least recently used* (LRU). The buffer *perfect* is used to mean perfect profile information. The *execution frequency* is the number of times the *store-load* pair was actually executed. For example, the execution frequency bin of 100 contains the *store-load* pairs that were executed from a 100 times to 999 times, the bin for 1000 contains *store-load* pairs executed from a 1000 times to 9999 times. The data presented is the number of *store-load* pairs *correctly* captured by each configuration of the memory profile buffer. *Store-load* pairs that are captured but for which the profile information indicates a different conflict frequency bin as compared to the *perfect* information are not shown.

The data in Table 5.1 indicates that most of the *store-load* pairs that conflict more than 80% of their time of execution are captured by the memory profile buffer. The sampled *store-load* pairs missing from the 80-100% row in the table are either deduced by the buffer to be in a lower bin or never captured by the buffer. This is also true for the other bins. Associativity in the memory profile buffer does not have much effect at an interrupt rate of 100,000 cycles used for the results in Table 5.1. It could, however, have a larger effect when the interrupt rate is increased. Using a replacement algorithm other than *least recently used* may also improve the accuracy

Table 5.1: 099.go

Conflict Frequency	Buffer Configuration	Execution Frequency					
		1	10	100	1000	10000	100000
0-39%	8x8	56	77	182	272	98	12
	16x4	56	78	182	272	98	12
	32x2	56	78	184	271	98	12
	64x1	56	78	183	271	98	12
	Perfect	56	79	184	276	98	12
40-59%	8x8	0	0	2	2	1	0
	16x4	0	0	1	2	1	0
	32x2	0	0	1	2	1	0
	64x1	0	0	2	2	1	0
	Perfect	0	0	2	3	1	0
60-79%	8x8	0	0	0	0	0	0
	16x4	0	0	0	0	0	0
	32x2	0	0	1	0	0	0
	64x1	0	0	1	0	0	0
	Perfect	0	1	3	2	0	0
80-100%	8x8	0	9	23	5	1	0
	16x4	0	9	23	5	1	0
	32x2	0	11	23	5	1	0
	64x1	0	9	22	5	1	0
	Perfect	0	12	39	12	9	2

of an associative memory profile buffer. One candidate replacement algorithm is *least executed*. According to this policy, the candidate for replacement would be the store that has the least execution count. Further exploration into the area of using the memory profile buffer and other hardware for the collection of profile information for memory disambiguation is required and is suggested as future work.

5.4 Summary

This chapter has dealt with using hardware-based profiling methods for memory disambiguation. Additional parallelism may be found in programs by being able to distinguish between the locations accessed by the *store* and *load* instructions in a program. Profiling may be used for this purpose. Custom hardware may be used to collect information about the memory locations accessed by the memory operations in a program during its execution. This information may then be fed back into the compiler to enhance the amount of optimization possible.

The memory profile buffer, the hardware structure for collecting profile information has been suggested in this chapter. Modifications to this buffer to reduce contentions have also been suggested. The instruction set architecture itself can be modified such that the memory profile buffer does not need to profile every memory operation. The memory operations that could result in some benefit in optimization are marked for profiling by the compiler by sing a bit in the encoding. This further reduces the burden on the memory profile buffer.

Preliminary results measuring the accuracy of the profile buffer have been presented. However, more work is required in the area using hardware-based profiling for memory disambiguation purposes. Future work will more thoroughly examine the efficacy of the hardware-based profiling methods in collecting memory dependence profile information and the use of such information in program optimization.

Chapter 6

Conclusions and Future Work

This thesis has examined fast and non-intrusive methods for collection of profile information in hardware. Many commercial microprocessors, such as the Pentium series [32] and the PowerPC 604 [33], incorporate some form of branch handling hardware. Such existing branch handling hardware, along with OS support, can be employed to obtain reliable profile information with imperceptible slowdown (0.4%–4.6%). This allows profiling of an application deployed in the field (e.g., during beta-testing). The application can later be retrieved for profile-based recompilation. The contents of the branch prediction hardware maintain information about the branches in code. This thesis has demonstrated the use of this information for obtaining profiles of the branches in a program. The effectiveness of this kind of information when used in compiler optimizations has also been investigated by applying such information to a prominent optimization, namely superblock scheduling. Methods of profiling for

both one-level and two-level branch predictors, and the quality of information they yield have been presented.

Some profiling methods today utilize the hardware support for debugging to provide limited profiling capabilities. For example, the PA-RISC architecture provides for a performance monitoring coprocessor that may be employed for collecting profile data [34]. These techniques require frequent interrupts to be able to gather moderately accurate data. The information collected by such methods is inadequate for the requirements of a profile-driven optimizing compiler. The use of custom hardware to collect profile information has been suggested here. The *profile buffer*, hardware whose mainline purpose is profiling, was presented. The profile buffer may be used when branch predictors are absent in a processor, or when the overhead for downloading information from a large predictor table is time-consuming. The branch predictor is far smaller than conventional branch predictor tables, thus enabling the collection of profile information with minimal overhead. Methods to obtain accurate profile information from the profile buffer have been presented. The information obtained from the profile buffer was used in superblock scheduling experiments to verify the accuracy and usefulness of the information.

Statistical theory was used to show that hardware-based profiling yields near-accurate information. Some issues in trace formation using profile information were dealt with. A method of using profile information collected using sampling methods such as in hardware-based profiling in trace formation algorithms has been suggested.

Traces may also be formed by profiling the paths in a program. A hardware structure known as the *history queue* was presented. This structure can be used to profile path traversals within a program. These paths themselves can then be treated as traces.

Memory disambiguation is another application for profile information. Static analysis of aliasing memory operations during compile time is difficult and time-consuming. Software-based profiling for memory disambiguation can be tedious. The thesis has suggested hardware structures for profiling memory operations. Preliminary results for the information that can be obtained from such structures have been presented.

The area of profiling and the use of profile information is in it's nascent stages. Detailed research into other profiling techniques and the use of profile information is recommended. Future work will involve research into profiling techniques for all kinds of profile information. This thesis has suggested hardware-based profiling techniques for obtaining control information and memory reference profiling. The application and efficacy of memory dependence profiling has not been addressed in this thesis and is a topic for future work. Many different profiles are possible for a program. Memory latency optimizations, for example, require a profile different from a control profile or a memory reference profile presented in this thesis. Profiling techniques for such profiles have not been addressed in the literature. Future work will investigate the use of hardware structures to efficiently extract such information from a program's execution.

Bibliography

[1] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," in *Proc. 1982 SIGPLAN Symp. Compiler Construction*, pp. 120–126, June 1982.

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

[3] 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.

[4] 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.

[5] S. McFarling and J. L. Hennessy, "Reducing the cost of branches," in *Proc. 13th Ann. Int'l Symp. Computer Architecture*, (Tokyo, Japan), pp. 396–403, June 1986.

[6] 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.

[7] 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* [48], pp. 247–255.

[8] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. M. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 138–149, May 1995.

[9] 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.

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

[11] J. S. Cox, D. P. Howell, and T. M. Conte, "Commercializing profile-driven optimization," in *Proc. 28th Hawaii Int'l. Conf. on System Sciences*, vol. 1, (Maui, HI), pp. 221–228, Jan. 1995.

[12] T. M. Conte, B. A. Patel, and J. S. Cox, "Using branch handling hardware to support profile-driven optimization," in *Proc. 27th Ann. Int'l Symp. on Microarchitecture*, (San Jose, CA), Nov. 1994.

[13] 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.

[14] 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.

[15] 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.

[16] M. Smith, "Tracing with pixie," Tech. Rep. CSL-TR-91-497, Center for Integrated Systems, Stanford University, Nov. 1991.

[17] J. Larus and T. Ball, "Rewriting executable files to measure program behavior," *Software Practice & Experience*, vol. 24, pp. 197–218, Feb. 1994.

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

[19] H. A. Rizvi, J. B. Sinclair, and J. R. Jump, "Execution-driven simulation of a superscalar processor," in *Proc. 27th Hawaii Int'l. Conf. on System Sciences*, (Maui, HI), 1994.

[20] D. W. Wall and M. L. Powell, "The Mahler experience: Using an intermediate language as the machine description," in *Proc. Second Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, (Palo Alto, CA), pp. 100–104, Oct. 1987.

[21] J. R. Ellis, *Bulldog: A compiler for VLIW architectures.* Cambridge, MA: The MIT Press, 1986.

[22] 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.

[23] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[24] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *Computer*, vol. 22, pp. 12–35, Jan. 1989.

[25] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the Hyperblock," in *Proc. 25th Ann. Int'l. Symp. on Microarchitecture*, (Portland, OR), pp. 45–54, Dec. 1992.

[26] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, "Reverse if-conversion," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, (Albuquerque, NM), pp. 290–299, June 1993.

[27] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta, "Predictability of load/store instruction latencies," in *Proc. 26th Ann. Int'l. Symp. on Microarchitecture*, (Austin, TX), pp. 139–152, Dec. 1993.

[28] MIPS Computer Systems, *UMIPS-V Reference Manual*. Sunnyvale, CA, 1990.

[29] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," in *Proceedings of the ACM SIGPLAN '92 Conference on Principles of Programming Languages*, pp. 59–70, 1992.

[30] A. D. Samples, *Profile-Driven Compilation*. PhD thesis, Computer Science Division, University of California, Berkeley, California, Apr. 1991. Report No. UCB/CSD 91/627.

[31] D. E. Knuth and F. R. Stevenson, "Optimal measurement points for program frequency counts," *BIT*, vol. 13, pp. 313–322, 1973.

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

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

[34] Hewlett-Packard, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. California, USA, 1994.

[35] T. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. 24th Ann. Int'l Symp. on Microarchitecture*, (Albuquerque, NM), pp. 51–61, Nov. 1991.

[36] T. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proc. 20th Ann. Int'l Symp. Computer Architecture*, (Ann Arbor, Michigan), pp. 257–266, May 1993.

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

[38] B. Calder and D. Grunwald, "Fast & accurate instruction fetch and branch prediction," in *Proc. 21st Ann. Int'l Symp. on Computer Architecture*, (Chicago, IL), pp. 2–11, Apr. 1994.

[39] B. A. Patel, "The effects of branch handling on superscalar performance," Master's thesis, Department of Electrical and Computer Engineering, University of South Carolina, Columbia, SC, 1995.

[40] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proc. 18th Ann. Int'l Symp. Computer Architecture* [49], pp. 266–275.

[41] T. M. Conte, B. A. Patel, K. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *Internation Journal of Parallel Programming*, vol. 24, Feb. 1996.

[42] S. Weiss and J. E. Smith, *POWER and PowerPC.* San Francisco, CA: Morgan Kaufmann, 1994.

[43] V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.

[44] J. C. H. McCall, *Sampling and statistics handbook for research.* Ames, Iowa: Iowa State University Press, 1982.

[45] G. T. Henry, *Practical sampling.* Newbury Park, CA: Sage Publications, 1990.

[46] C. R. Blyth and H. Still, "Binomial confidence intervals," *Journal of the American Statistical Assoc.*, vol. 78, pp. 108–116, 1983.

[47] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Trans. Comput.*, vol. 38, pp. 663–678, May 1989.

[48] *Proc. 26th Ann. Int'l Symp. on Microarchitecture*, (Austin, TX), Dec. 1993.

[49] *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), May 1991.