Accurate and Practical Profile-Driven Compilation Using the Profile Buffer

Thomas M. Conte Kishore N. Menezes Mary Ann Hirsch
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina 27695
{conte, knmeneze, mahirsch}@eos.ncsu.edu

Abstract

Profiling is a technique of gathering program statistics in order to aid program optimization. In particular, it is an essential component of compiler optimization for the extraction of instruction–level parallelism. Code instrumentation has been the most popular method of profiling. However, real-time, interactive, and transaction processing applications suffer from the high execution-time overhead imposed by software instrumentation. This paper suggests the use of hardware dedicated to the task of profiling. The hardware proposed consists of a set of counters, the profile buffer. A profile collection method that combines the use of hardware, the compiler and operating system support is described. Three methods for profile buffer indexing, address-mapping, selective indexing, and compiler indexing are presented that allow this approach to produce accurate profiling information with very little execution slowdown. The profile information obtained is applied to a prominent compiler optimization, namely superblock scheduling. The resulting instruction-level parallelism approaches that obtained through the use of perfect profile information.

1 Introduction

Profile information is steadily gaining importance in the optimization of program execution, especially in the areas of hand–tuning of programs [12], trace scheduling [11], superblock scheduling [6], data preloading [7], branch prediction [19], and improved instruction cache performance [6]. 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 [4], [6], [13], [11], [18]. In one example, traditional optimizations when combined with profile-driven superblock formation enhance execution performance by an additional 15% [6]. However, profile collection has severe drawbacks that have limited its commercial use.

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. Next, this instrumented code is executed with a variety of batch inputs. After these multiple executions, the program statistics are calculated based on the profiling 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* [20], inserts about five instructions in every basic block [3]. Ball and Larus measured the slowdown of *pixie* required for arc–based profiling as between 1.11 to 5.24 times [3]. 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 [3], [4], [21], [17]. 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%) [10]. Code instrumentation techniques, therefore, may not be appropriate for real–time, interactive, and transaction processing applications.

If profiling is to gain commercial acceptance, profiling 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 by Conte, *et al.* [8] 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 [8]. 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 [10].

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 [14]. 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.

Although hardware-based profiling achieves very little slowdown, it provides less than ideal profiles [8],[9]. This is due to the small size of the branch prediction hardware [8] and the relatively coarse-grain nature of current branch prediction state machines. In this paper, the *profile buffer* is proposed for the collection of profile information that is similar in slowdown to branch predictor based profiling. 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 [6]. The performance measurement data based on the execution of the optimized program indicates that a small buffer (16–64 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 paper. The following section discusses Conte, *et al.*'s previous method of collecting profile information using the branch–handling hardware in a processor [8]. In addition, 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 2.2–2.4. 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. Section 3 presents a summary of experimental results and discusses the tradeoffs between the various schemes. The paper concludes with comments and suggestions for future work.

2 Profiling in Hardware

The insertion of additional instructions required by dynamic profiling through software causes significant execution overhead. The use of efficient hardware schemes that utilize existing branch handling hardware with OS support to obtain profile information was proposed by Conte, *et al.* [8]. For example, the *two-level predictor* profiling scheme uses Yeh and Patt's PAs adaptive training branch predictor [23]. Briefly, one buffer in PAs, the *history register table* (HRT), contains histories for dynamically executing branch instructions. This buffer is referenced by the instruction address of each branch. The second buffer, *pattern table* (PT), contains a state machine for predicting the outcome of a branch [23]. When a branch access occurs, the HRT is referenced. The history obtained from the HRT is used to index into the PT to calculate a branch prediction. The actual outcome of the branch after execution is later used to update both the history and the state machine.

The profiling scheme for PAs periodically dumps the HRT to obtain the profile of the entire program. However, the limited size of each history register in the HRT (10 bits in [8]) results in sparse profile information being downloaded. Contentions due to the limited size of the HRT (1024 entries in [8]) also introduce error. The combined sampling error is reduced slightly by using a *marker* bit in each history register to mark the boundary between valid and invalid information [8],[9]. But the overall lower quality profile information collected when compared against traditional code instrumentation profiling is the main drawback of the scheme.

2.1 The profile buffer

In an effort to obtain near-accurate profile information and simultaneously maintain a low profiling overhead, a specialized buffer called the *profile buffer* is proposed. This buffer is illustrated in Figure 1. 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 are updated based on whether the branch was taken.

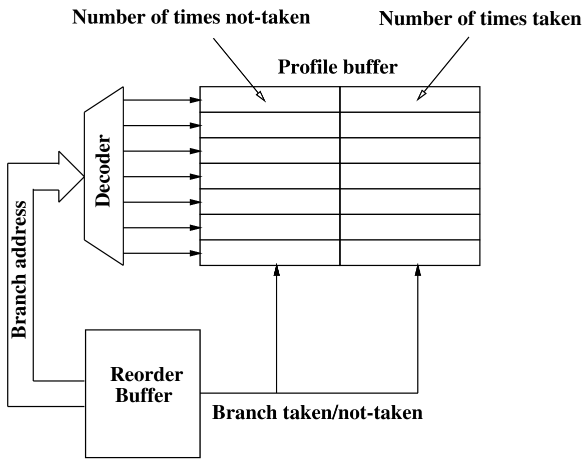

Figure 1. The profile buffer. (Entries in the buffer are counters).

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. Alternatively, the operating system may generate a timer interrupt and perform the required action. Because the updating of the information in the *profile buffer* is completely program-controlled and does not require intervention by the operating system, the storage back to memory constitutes the only slowdown possible. 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 [8]. 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 [8]. The profile buffer is dumped out in a manner similar to branch handling hardware but this buffer is much smaller (e.g., 32 entries). Therefore, the schemes proposed in this paper have a slowdown comparable to or lower than the average of 1.02 times. 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.

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 paper, 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 inaccuracy caused due to contentions is measured by using the *arc error* metric. Assume,

$$\Delta e_i = \mid e_{ij} - \hat{e}_{ij} \mid \qquad (1)$$

where, $\hat{e}_{ij}$ is the frequency of the edge from i to j in the measured profile, e_{ij} is the frequency in the *true* profile. Then, the distribution function is given by,

$$f(E) = \sum_{i_{s.t.} W_i = W} \Delta e_i \qquad (2)$$

where, W_i is the execution weight of block i, and W is an execution weight category (e.g. execution weight of 100–1000).

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* [13]). The profile information collected using the profile buffer scheme was applied to a superblock formation algorithm implemented in the University of Illinois IMPACT compiler [5],[15]. The execution rate of useful instructions retired per cycle (IPC) of the profile-based scheduled code was used as the metric of comparison. All experiments were simulated on an eight–issue machine with uniform functional units. As a standard for all tables that contain IPC information, the *exact* column indicates the IPC obtained through the use of accurate profile information obtained through code instrumentation and superblock scheduling. The column *none* shows the IPC for the benchmarks compiled without any instrumentation for profiling (i.e., basic-block only scheduling).

The following three sections present the performance of the profile buffer of different sizes and indexing schemes. The three indexing schemes that were reviewed, address mapping, selective indexing, and compiler indexing, are described below.

Table 1. The performance (IPC) of code compiled using profiling results obtained from buffers that employ the *address mapping* scheme.

Benchmark	Instructions/cycle					
	exact	8	16	32	64	*none*
compress	2.71	1.66	1.64	1.64	1.63	1.38
espresso	2.36	1.36	1.36	1.35	1.36	1.20
eqntott	1.86	1.24	1.24	1.24	1.24	1.01
gcc	1.83	1.35	1.36	1.39	1.39	1.27
li	1.65	1.24	1.25	1.26	1.26	1.15
sc	2.29	1.33	1.39	1.35	1.37	1.25
harmonic mean	2.06	1.35	1.36	1.36	1.36	1.20

Table 2. Contentions as percentage of total accesses for *address mapping*.

Benchmark	Percent Contentions			
	8	16	32	64
compress	41.3%	26.0%	12.6%	7.9%
espresso	29.7%	19.1%	13.4%	5.2%
eqntott	6.1 %	3.7 %	1.8 %	0.5%
gcc	63.4%	55.4%	48.3%	40.9%
li	74.3%	68.1%	55.3%	41.6%
sc	66.4%	45.8%	28.0%	27.2%
mean	46.9%	36.3%	26.6%	20.6%

2.2 Address Mapping

The first approach to indexing the profile buffer is *address mapping*. This simple method uses the branch instruction address to directly reference the profile buffer. It is identical to indexing schemes used in branch prediction hardware, and its performance is comparable to that presented in [8],[9][1]. 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 1. A distinct performance increase is seen through the use of profile information collected using the profile buffer with *address mapping*. However, the performance levels achieved fall short of those possible with exact profile information (*exact* in Table 1). This performance degradation is due to an abundance of contentions when simple *address mapping* is employed. The contentions for entries in the profile buffer are shown in Table 2. 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. The effect of the contentions is seen in Figure 2. The *arc error* reduces as the size of the buffer is increased, but is still unacceptably high. These results indicate that reducing the number of contentions should be a priority for obtaining better profile information. Traditionally, branch target buffers (BTB) that have similar properties as the profile buffer have overcome the problem by employing larger buffers (512 or 1024 entry) [1], [22], 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 sections.

[1]The performance of address mapping is slightly higher than that of the PAs profiling of [8],[9] because the profiling buffer employs two, 16 bit counters instead of 10 bit history registers. However, because it is the lowest performance of the three indexing schemes presented here, it also serves to represent the upper bound of performance for PAs profiling from [9] for the purposes of this study.

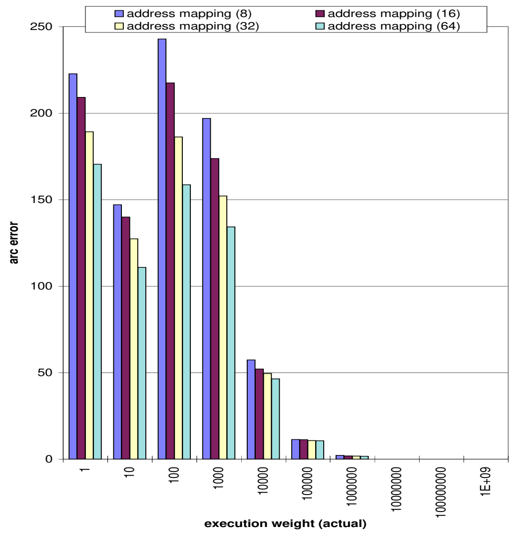

Figure 2. Arc error for the address mapping technique vs. the actual execution weights of the branches.

2.3 Selective indexing

A control flow graph (CFG) is a directed graph that uses basic blocks as nodes and control transfers as arcs. A CFG that will serve as a running example is given in Figure 3. The *arc-based profile* is a method of profiling where code is added to record the flow of execution along all block-to-block transitions (arcs) in the CFG [2]. This simple implementation of *arc-based profiling*, however, requires that the frequencies of all arcs be calculated. When using a profile buffer, this implies that every branch is tabulated in the profile buffer, thereby causing an excessive 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 [3], [17], [21]. Knuth and Stevenson [17] describe a method for reduction in the number of nodes that need to be probed. Their algorithm is based on Kirchoff's first law, which states that the flow into a vertex equals the flow out of that vertex. Therefore, if there are n vertices and m arcs in a CFG, then only $m - n - 1$ arc flow frequencies need to be measured in order to determine the frequencies of all the arcs in the original CFG. The Knuth-Stevenson algorithm is best illustrated by an example. A detailed example of this algorithm is given in Figures 3–6. The arcs represent possible flows of control as the algorithm is performed.

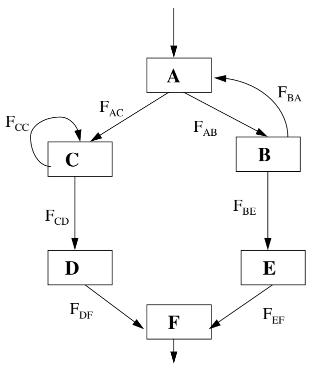

Figure 3. An example CFG.

The *selective indexing* profiling scheme uses the Knuth-Stevenson algorithm to determine the branches that should access the profile buffer. The profile buffer uses an arc-based profiling scheme because it endeavors to measure the frequencies of the arcs. However, the instrument of measurement, namely the branch instruction, exists in a node. Therefore, the Knuth-Stevenson algorithm can be directly applied to reduce the number of branches that need to be profiled, consequently reducing the number of accesses to the profile buffer and the number of contentions. 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

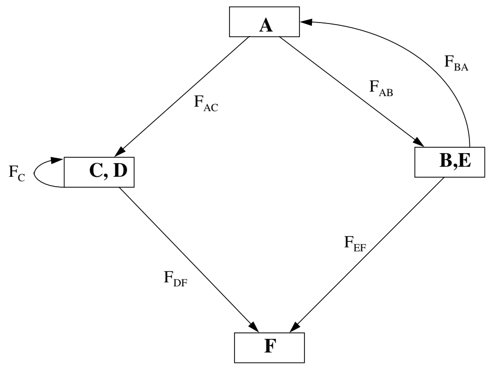

Figure 4. The first step in the Knuth-Stevenson reduction procedure. In this step, multiple vertices will be merged into a single combined vertex. For example, because $F_C = F_{CC} + F_{CD}$, we can make F_C by merging vertices, C and D.

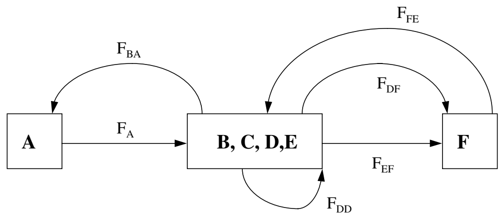

Figure 5. The second step in the Knuth-Stevenson reduction procedure. In this step, nodes are merged together so the number of vertices can be reduced further. For instance, F_{AC} and F_{AB} are merged into F_A.

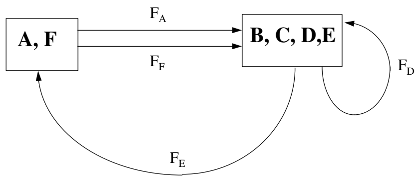

Figure 6. The Knuth-Stevenson reduced form of CFG. This CFG is produced by recursively applying step 2 until the CFG is in an optimal form. The nodes, A and E, are merged together so F_F is formed from F_{DF} and F_{EF}.

Table 3. Effect of reduction in profiling branches using *selective indexing*. (Contentions are shown as percentage of original accesses).

Benchmark	Change in accesses	Percent Contentions			
		8	16	32	64
compress	-46.2%	20.2%	12.1%	3.5%	0.5%
espresso	-30.5%	27.0%	10.4%	5.5%	2.1%
eqntott	-39.1%	34.1%	0.8%	0.1%	0.0%
gcc	-42.6%	28.1%	22.5%	17.2%	12.0%
li	-51.6%	30.6%	26.7%	20.1%	14.3%
sc	-43.1%	18.8%	6.7%	1.4%	0.3%
mean	-42.2%	26.5%	13.2%	8.0%	4.9%

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 could be changed to BEQ during code generation. Another alternative for architectures that use predicated branches (such as *HPL PlayDoh* [16]), 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.

The results for the *selective indexing* scheme using the suggested ISA change are presented in Table 3. 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 40%. *Selective indexing* also affects the number of contentions across the buffer sizes. The percentage of contentions for *gcc* in an eight-entry buffer is 28.1% compared with 63.4% for *address mapping*. For 64 entries,

Table 4. The performance (IPC) of code compiled using profiling results obtained from buffers that employ the *selective indexing* scheme.

Benchmark	Instructions/cycle					
	exact	8	16	32	64	*none*
compress	2.71	2.32	2.78	2.45	2.50	1.38
espresso	2.36	2.05	1.93	2.18	2.29	1.20
eqntott	1.86	1.86	1.98	1.86	1.86	1.01
gcc	1.83	1.60	1.66	1.72	1.74	1.27
li	1.65	1.36	1.40	1.51	1.49	1.15
sc	2.29	1.71	2.00	2.20	2.10	1.25
harmonic mean	2.06	1.76	1.88	1.93	1.94	1.20

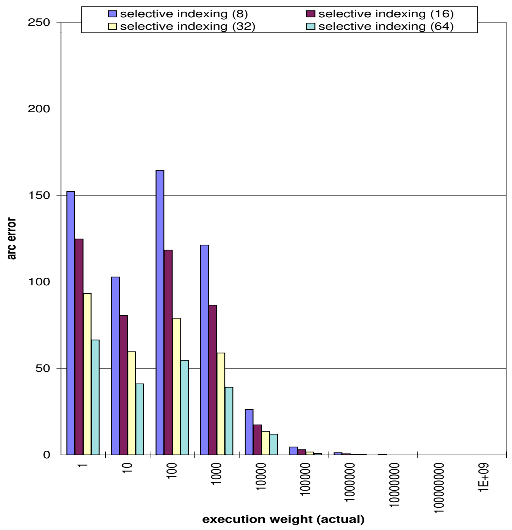

Figure 7. Arc error for the selective indexing technique.

the percentage of contentions for *gcc* is reduced to 12% from 40.9%. In addition, *li* also has a significant decrease in the percentage of contentions in an eight-entry buffer, 30.6% compared with 74.3% for *address mapping*. These reductions in contentions are due to the decrease in the number of branches competing for the same entries in the buffer.

Figure 7 shows the dramatic reduction in *arc error* resulting from the reduction in contentions. The error for branches that execute from a 100–1000 times is reduced by approximately 32% when compared to the address mapping technique for the eight–entry buffer. The reduction in error is seen across all buffer sizes. As seen in Table 4, the higher accuracy of the profile information assists in providing better performance (IPC). The results for superblock scheduling using the information obtained through the *selective indexing* approach in general are comparable to those obtained with *exact* profile information. However, in the case of *espresso*, a 16-entry buffer produces worse results than the eight-entry buffer. This is primarily due to erroneous profile information which causes the superblock formation algorithm to make incorrect decisions. Because the number of instructions in a basic block vary from one block to another, the addresses of the branch instructions do not follow any fixed pattern. This situation could lead to adjacent branches or branches in close proximity to access the same location in the buffer, thereby causing error in the profile information obtained. The error, however, can be reduced by controlling the buffer access pattern (shown in the section 2.4).

The performance for the benchmark *compress* in conjunction with *selective indexing* in some cases produces performance results that are better than their *exact* counterpart. This phenomenon was discussed in [9], where code scheduled for benchmarks such as *compress* were examined by hand. Two explanations were discovered: (*i*) erroneous profiles produce better performance because the error resulted in a larger superblock, which in turn produces a larger scope for the scheduler; and, (*ii*) there were benefits for scheduling down the less-likely path when the difference between arc counts was small. As an example of the latter, several cases were found where the error changed the superblock selection from the not-taken path of the branch to the taken path. Although the taken path was not executed as frequently as the not-taken path, the scheduler found many more opportunities for code motion along the less-likely path, resulting in higher overall performance. This evidence supports the fact that this phenomenon is due to a slight non-optimality caused by decoupling superblock formation from scheduling, rather than a shortcoming of hardware-based profiling [9].

2.4 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 branches for profiling as guided by the Knuth–Stevenson algorithm. It then assigns indexes to the selected branches in such a way as to guarantee that adjacent or near-by branches do not map into the same entry.

The efficiency of the *compiler indexing* scheme is illustrated in Table 5. The further reduction in 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 number of contentions for *gcc* is reduced to 26.8% (eight entries) as compared with 28.1% for *selective indexing*. Therefore the profile information obtained through this support from the compiler is expected to be more accurate and consistent.

The *arc error* for the *compiler indexing* scheme as shown

Table 5. Percent contentions of original accesses after reduction in profiling branches and using *compiler indexing*.

Benchmark	Percent Contentions			
	8	16	32	64
compress	15.5%	6.3%	0.0%	0.0%
espresso	11.8%	3.6%	0.9%	0.2%
eqntott	1.5%	0.2%	0.2%	0.2%
gcc	26.8%	22.3%	16.9%	11.1%
li	30.4%	25.5%	20.2%	15.6%
sc	10.2%	7.8%	0.5%	0.3%
mean	16.01%	10.96%	6.45%	4.56%

Table 6. The performance (IPC) of code compiled using profiling results obtained from buffers that employ the *compiler indexing* scheme.

Benchmark	Instructions/cycle					
	exact	8	16	32	64	*none*
compress	2.71	2.44	2.37	2.71	2.71	1.38
espresso	2.36	2.17	2.21	2.32	2.35	1.20
eqntott	1.86	1.86	1.67	1.86	1.86	1.01
gcc	1.83	1.49	1.59	1.68	1.71	1.27
li	1.65	1.37	1.41	1.46	1.52	1.15
sc	2.29	2.07	2.10	2.18	2.21	1.25
harmonic mean	2.06	1.82	1.83	1.95	1.98	1.20

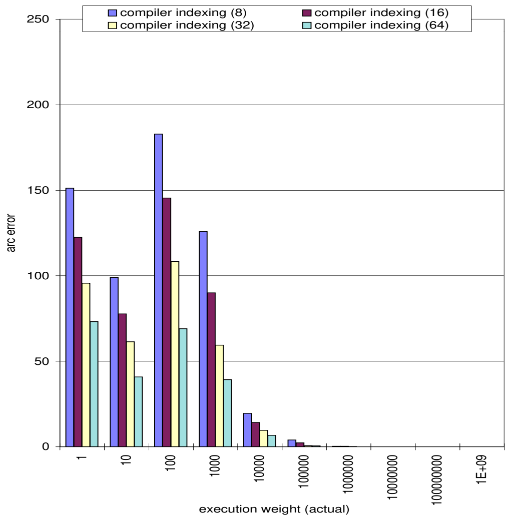

Figure 8. Arc error for the compiler indexing technique.

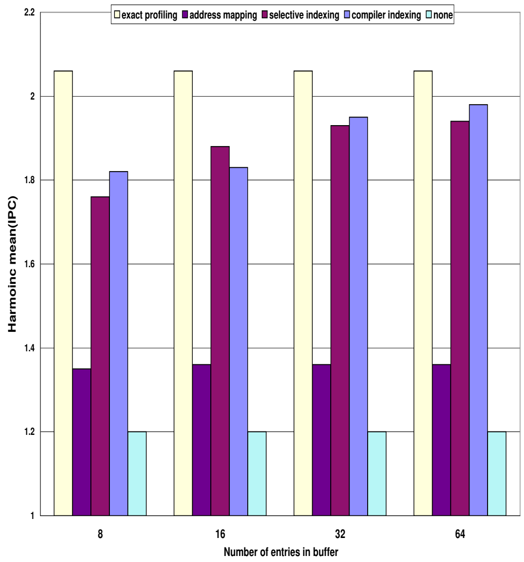

Figure 9. Comparison of the profile buffer schemes.

in Figure 8 is much lower that that for the *address mapping* technique and comparable to that for *selective indexing*. The technique yields lower errors than *selective indexing* for the most-frequently executed branches (i.e., execution weights 10^5–10^9). Table 6 shows the results of superblock formation obtained using the profile information from *compiler indexing*. The harmonic mean throughput is now seen to consistently increase as the size of the buffer is increased, demonstrating the increase in the accuracy of the profile information due to the reduction in contentions. The IPC achieved for a 32-entry and 64-entry buffer even approaches that obtained by using *exact* profile information. In the case of an eight-entry or 16-entry buffer, the IPC achieved is far better than without the use of any profile-information. However, the IPC prediction for the benchmark, *gcc*, is less accurate when *compiler indexing* is used. This is due to the fact that benchmarks such as *gcc* and *li* have a large number of function calls causing branches across functions to conflict. This could be remedied by inter–procedural index assignment or hardware *profile buffer* windows (discussed further below).

3 Analysis

The performance of the profile buffer is dependent on its size and its indexing scheme. This section presents some analysis of the results from the previous sections. The *harmonic mean* of the IPC is used in the performance analysis that follows, because the performance of the profile buffer schemes is expressed as a rate (IPC). The performance for all the profile buffer indexing schemes, *exact* profiling, and no profiling (*none*) are shown in Figure 9. It can be seen that the simple *address mapping* scheme provides a 13% execution performance increase when compared with the no profiling version. However, this scheme only provides about 78% of the possible peak performance. This failure to deliver peak performance is caused by the incompleteness of the information the *address mapping* scheme provides to the compiler due to the excessive number of contentions. In this scheme, there is almost one contention for every two access to the profile buffer (from Table 2).

The other two schemes, *selective indexing* and *compiler indexing*, show significant increases in performance when compared against no profiling and *address mapping*. *Selective indexing* achieves 85% – 94% of the performance of *exact* profiling, while *compiler indexing* provides a slightly higher 88% – 96% of the peak performance. The *selective indexing* approach performs almost as well as the *compiler indexing* method because *compiler indexing* only approximates *selective indexing* for workloads that use many function calls, such as *li* and *gcc*. This effect is more prominent when the buffer size is small (such as an eight or 16-entry buffer). Figure 10 shows a comparison in *arc error* between *selective indexing* and *compiler indexing* for the branches with high execution weights. In the case of such branches, *compiler indexing* yields lower error since the method of indexing limits the number of contentions when branches in close proximity are executed frequently, such as in tight inner loops. Figure 11 compares the arc error between the two techniques for lightly and medium executed branches. *Se-*

lective indexing provides comparable and sometimes more accurate information for such branches. This is a result of the naive implementation of compiler indexing which suffers when programs make frequent function calls as in *gcc* and *li*. The higher accuracy for medium-executed branches using *selective indexing* results in performance comparable to *compiler indexing* after superblock formation. At larger buffer sizes, the *compiler indexing* method can be expected to do better than *selective indexing* (see Figure 9). This suggests that integrating detailed inter-procedural analysis into the index allocation algorithm would improve *compiler indexing*, allowing for buffer sizes less than 32 entries.

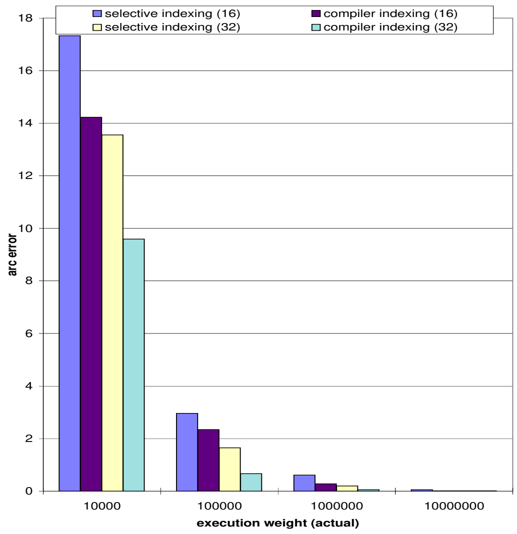

Figure 10. Comparison of indexing techniques for blocks with high execution weights.

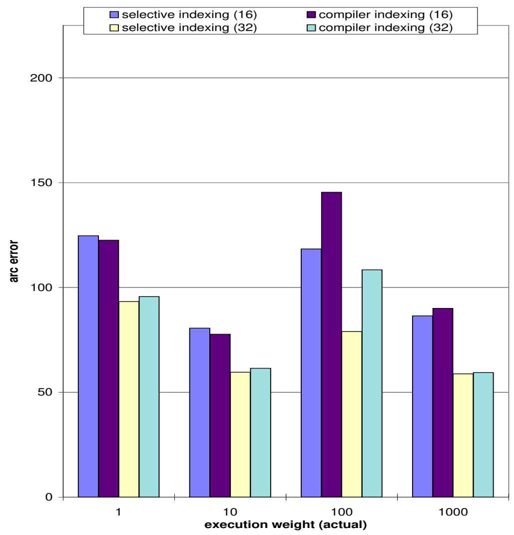

Figure 11. Comparison of indexing techniques for blocks with low execution weights.

4 Conclusion

A large amount of research in recent years has concentrated on the exploitation of instruction–level parallelism through automated compiler optimization using information obtained from previous runs of a program [11],[6],[7],[19],[6]. The techniques of gathering the required profile information, however, have depended upon traditional code instrumentation methods, which are not suitable for real–time, operating system, or interactive workloads, due primarily to the slowdown software profiling introduces. Fast and non-intrusive methods are required for effective, efficient profiling in order to gain commercial acceptance of profiling.

This paper further developed the ideas presented by Conte, *et al.* [8] 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. 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 possibly access the *profile buffer*. The branches that access the buffer were determined using the Knuth-Stevenson algorithm. 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* that is assigned at compile time.

The *address mapping* and *selective indexing* schemes are easily applicable to existing architectures without much overhead in cost. For *selective indexing*, one bit in the branch instruction is required to indicate the branch to be profiled. It is also possible to implement *selective indexing* without change 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*. The dumping of the buffer 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, 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 formation, 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* perform well. The experimental results further indicate that a buffer with 32 or more entries provides about 95% of the performance that is achieved with the use of the exact multiple pass profiling sequence. 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 profiling buffer becomes a viable and commercial technique to aid profile-driven optimization.

Acknowledgments

This work was supported by Intel Corporation and the National Science Foundation under grants MIP-9696010 and MIP-9625007.

We especially thank the University of Illinois IMPACT group for use of the IMPACT compiler.

References

[1] D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. *IEEE Micro*, 13(3):11–21, June 1993.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. Technical Report 1031, Computer Sciences Dept., University of Wisconsin-Madison, 1991.

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

[4] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, Albuquerque, NM, June 1993.

[5] 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. International Symposium Computer Architecture*, pages 266–275, Toronto, Canada, May 1991.

[6] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software–Practice and Experience*, 21(12):1301–1321, Dec. 1991.

[7] 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.

[8] T. M. Conte, B. A. Patel, and J. S. Cox. Using branch handling hardware to support profile-driven optimization. In *Proc. 27th Ann. International Symposium on Microarchitecture*, San Jose, CA, Nov. 1994.

[9] 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*, 24(2), Feb. 1996.

[10] J. S. Cox, D. P. Howell, and T. M. Conte. Commercializing profile-driven optimization. In *Proc. 28th Hawaii Int'l. Conf. on System Sciences*, volume 1, Maui, HI, Jan. 1995.

[11] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, July 1981.

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

[13] 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*, pages 247–255, Austin, TX, Dec. 1993.

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

[15] 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*, 7(1):229–248, Jan. 1993.

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

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

[18] 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. International Symposium Computer Architecture*, pages 138–149, Santa Margherita Ligure, Italy, June 1995.

[19] S. McFarling and J. L. Hennessy. Reducing the cost of branches. In *Proc. 13th Ann. International Symposium Computer Architecture*, pages 396–403, Tokyo, Japan, June 1986.

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

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

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

[23] T. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proc. 24th Ann. International Symposium on Microarchitecture*, pages 51–61, Albuquerque, NM, Nov. 1991.