

Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization

Thomas M. Conte* Burzin A. Patel[†]
Kishore N. Menezes* J. Stan Cox[†]

*Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina 27695-7911

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

Abstract

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

Keywords: Branch handling, profiling, compiler optimization, microarchitecture

1 Introduction

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

A typical profiling method employs the following *compile-run-recompile* sequence: (1) the program is compiled with additional profiling instructions inserted within each basic block; (2) it is then run using several different test inputs; and (3) the resulting profile information is used to drive a profile-based recompilation of the original program. Although the final product is well-optimized code, the overhead associated with profiling is high (a profiling version of a program may experience a slowdown of 50% and to several hundred percent), which poses a problem for the collection of a representative set of inputs for interactive or real-time applications. This often renders profiling impractical for many applications use.

The technique of static branch behavior estimation solves some of the problems related to gathering profile data [6]. However, it is not as accurate as profiling [7],[8]. When used for superblock scheduling, for example, static estimates provide approximately half of the speedup that profiling can achieve [9]. Moreover, static estimation can only predict the

direction of branches, not the relative execution frequencies of the code itself. This latter information is used to limit the amount of code duplication caused by some optimizations. Without it, optimizations are susceptible to code size explosion.

This paper examines *hardware-based profiling*, a fast and unobtrusive method for the collection of profile information. Many commercial microprocessors, such as the Pentium series [10] and the PowerPC 604 [11], incorporate some form of branch handling hardware. This branch handling hardware, along with OS support, can be employed to obtain reliable profile information with minimal impact on program execution time (0.4%–4.6% slowdown). This fact allows profiling of an application deployed in the field (e.g., during beta-testing), which can later be retrieved for profile-based recompilation. Since real usage patterns are profiled, the method also solves the problem of obtaining valid test inputs for profiling. This approach also 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 make it a viable technique for use in program optimization.

The following section reviews common hardware branch prediction mechanisms that are used in the experiments presented in this paper. Methods for obtaining profile information from such hardware are discussed in the third section. Although these methods are less accurate than full-fledged profiling, they are significantly more accurate than static estimates. Metrics to measure error are discussed in Section 3.5. The fourth section presents experimental results and discusses the tradeoffs between the various schemes. The paper closes with a discussion of the uses for hardware-based profiling.

2 Contemporary 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 *branch target buffer* (BTB) entry, which contains a state machine for predicting the outcome of each branch, and the last target address used by the branch. (Some BTB implementations also store copies of the instructions from the last target address.) Each entry's state machine is used to predict a branch's behavior early in the processor pipeline, and the actual branch behavior is used to update the state after execution (depicted in Figure 1). The most common state machine for one-level schemes is the two-bit counter predictor [12], which is implemented in several contemporary processors, including the Intel Pentium [10] and the PowerPC 604 [11].

The size for the one-level BTB buffer affects the branch prediction accuracy since several branches may contend for the same entry in the buffer. The effect of the size of a buffer (S) on the number of contentions is examined and plotted in Figure 2. The knee for the majority of the curves occurs near the 512- to 1024-entry region. Buffer sizes larger than 1024 are able to provide entries for most of the branches in each benchmark, with the exception of the *gcc* benchmark, which has a relatively high number of contentions for all BTB sizes due to its large pool of static branches [13]. Taking these observations into consideration, a 1024 entry branch target buffer is selected for the extent of this study.

Two-level schemes use two separate buffers and function in a slightly different manner

from the one-level schemes (depicted in Figure 3). (These schemes have been studied extensively by Yeh and Patt [14],[15], and their nomenclature will be used here.) The first level buffer, which is termed the *history register table* (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 using the branch address, then using the history string to index into the *pattern table* (PT). The PT stores a state machine used to predict the branch (i.e., a two-bit counter as in the one-level scheme). This approach decouples the branch prediction from the address of the branch instruction, which dramatically improving branch prediction accuracy. Whereas a one-level, 1024-entry BTB achieves approximately 90% accuracy for the SPECint92 benchmarks, Yeh and Patt’s algorithm has been shown to achieve 96% accuracy [14],[15].

The following section presents methods to use branch prediction hardware for profile collection. Although these hardware schemes were not designed for this task, empirical results presented later in the paper demonstrate their usefulness for profile collection.

3 Using Branch Prediction Hardware for Profiling

In order to explain how branch handling hardware can be used for profiling, it is first important to review how profile information is used for profile-driven optimization. Such optimizations use a structure known as a *weighted control flow graph* (WCFG), which is a directed graph with basic-blocks as nodes and arcs due due to either code labels or branch instructions. Figure 4 presents an example of a WCFG. 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* [1], and the IMPACT project *superblocks* [2]. An example of superblock formation is shown in Figure 5. In the example, the profile weights are used to find code that tends to execute together (these larger block structures are where superblock formation gets its name). In the example, block D is duplicated to form block D' , allowing the formation of superblock C/D' . Chang, *et al.* reported a speedup of 15% when superblock formation was used in conjunction with traditional optimizations [2].

Profiling may be defined as the action of recording the arc and node weights for a WCFG. There are several methods for this. One method is *trace-based profiling*, where extra code is inserted at the beginning of each basic block to record the block identity during execution. The trace of block identities is then parsed to extract weights, either periodically during execution, or after program completion. Example trace-based profilers include *Spike* [16], *pixie* [17], and QPT [18]. Implementation decisions for this method can result in execution slowdown of 50% to several hundred percent.

A second profiling method is *arc-based profiling*, where code is added to record the flow of execution along all arcs in the WCFG [19]. Specifically, for each arc in the graph, the original destination of the arc is changed to point to a new transition block, and an unconditional branch to the original destination is added to the end of this new block. A table of counters for all arcs in the WCFG is added to the object code by the compiler, and instructions to increment table entries are placed inside each transition block. After execution, node weights

can be recovered from arc weights by summing all arc counts into or out of a basic block. Empirically, arc-based profiling results in a 50% slowdown [20].

This paper presents a third, *hardware-based profiling* approach that uses the contents of hardware branch prediction buffers to construct weights for the WCFG. The branch prediction hardware is sampled periodically using kernel-mode instructions which are implemented in several commercial processors. As discussed in the previous section, the hardware buffers hold branch 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. The specific procedure for recovering this information via hardware is as follows:

1. A program is compiled with a special identifier token (e.g., a Unix magic number), indicating that it contains a table of WCFG arcs, which is similar in structure to the table built for arc-based profiling. (During compilation, the code structure is also adjusted slightly, which is described below in Section 3.1.)
2. During execution, the kernel periodically reads the hardware branch prediction buffer and uses its contents to update the arc table. This updating is performed during each kernel entry (termed a *sampling point* in the discussion below).
3. On program exit, the arc table is updated on disk in a section of the executable file.

In order to measure the overhead associated with hardware-based profiling, the procedure was prototyped using a Pentium-based AT&T server system. The results are presented in

Table I. The results show very little difference in execution time between profiled programs and unmodified programs (0.4%–4.6%).

3.1 Code adjustments to support arc-based profiling

Some code adjustments are required to convert hardware branch information into arc weights. For example, indirect jumps can produce blocks with more than two outgoing arcs, reducing the one-to-one mapping between a buffer entry and an arc. When the target address is available in the buffer, this phenomenon can be tolerated. However, when it is not, steps need to be taken by the compiler to reduce the effects of this class of jumps.

The two primary sources of indirect jumps in C are due to call-through-pointers and *switch* statements. Call-through-pointers are not problematic, since trace selection is traditionally performed on a per-function basis. The impact of *switch* statements can be negated by conversion into a chain of *if* statements, which can be undone when the profiling information is used to recompile the program. The desirable effect of this conversion is an increased branch target address predictability.

The second adjustment concerns arcs that are not due to explicit branch instructions. One example is code labels for the cases of a C *switch* statement. During execution, control flow may pass from one logical node in the WCFG to another by crossing several label boundaries. This can also be termed fall-through code, since control flow falls from the end of one block to the beginning of another without encountering an intervening branch. These control flow arcs are never allocated entries in the hardware buffer since they have no

corresponding branch instruction. There are two possible solutions to this problem: (1) use the structure of the static graph to propagate profile information to these blocks, or (2) add a branch instruction to all fall-through paths that jump to the next sequential instruction. The first solution involves using the known arc weights to infer the arc weights of the fall-through code, and is preferable since the second solution reduces the performance of profiled code.

3.2 One-level profiling

Hardware-based profiling on machines that employ one-level schemes requires a mechanism to estimate the arc-weights. Although the BTB contains a counter, it is only a two-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 non-taken) further limits its practical weight estimation range to one bit. Instead of using the counter directly, the one-level 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 the performance hardware records N executed during a sampling period, and the average number of instructions per basic block, \bar{B} (from the compiler), and d entries in the BTB have been accessed at least once during the sample, then each BTB entry is accessed approximately $\hat{w} = N/(\bar{B} \times d)$ times. Some indicator of the validity of a branch history is required to determine d . This can be implemented using a *dirty bit* for each entry of the BTB, which requires no hardware modification if the BTB maintains a tag store.

This estimate assumes all branches are executed with the same frequency during a sampling period.

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. Based on this, arc counters corresponding to heavily-weighted branches are incremented by $(2\hat{w})$ at each sampling point, whereas time-varying branches are incremented by \hat{w} . This succeeds because many profile-driven optimizations (e.g., trace formation) depend on the relative weights between arcs rather than the absolute magnitude of the weights. The details of the translation are summarized in Table II. The error introduced by these approximations is discussed in Section 4 below.

3.3 Two-level profiling

Unlike the case of one-level profiling, the history saved in two-level branch predictors can be used to reconstruct arc weights directly at each sampling point. This is done by counting the number of 1's in each history register and dividing by the register width. Some slight modifications are needed to enhance the accuracy of the weights. For example, after a history register is sampled, it must be updated in some fashion so that its present contents are not

used again at the next sampling point. Since a buffer sampling point occurs at kernel entry points, there is little reason to preserve the contents of the history registers. The history registers are therefore reset after being sampled. This has a negligible effect on the prediction accuracy since context switches will normally result in a partial or near-total flush of the buffer.

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 [21], and the the best-performing was found to be the *marker bit* scheme. This scheme records the boundary between the valid and invalid branch histories, as illustrated in Figure 6. After each sampling point, the history registers are zeroed and the LSB of each register is set to ‘1’ (i.e., ‘00...01’). 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...01xxxx’, 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 marker-bit scheme only changes the way history register values map into pattern table entries, but does not result in higher contentions between entries in the pattern table. Due to this, the effect of the full bit on the accuracy of the two-level scheme was found to be negligible [21].

3.4 Node weight estimation

Trace selection and its related techniques rely not only on arc weights but also on the relative weights of the nodes (i.e., basic blocks) in the WCFG. These weights are used to select which nodes are the most-important for optimization (the heavily-weighted nodes), and in the case of superblock formation, those that should not be duplicated (the lightly-weighted nodes). Although hardware-based profiling reconstructs arc weights, reconstruction of node weights constitutes another problem. One way to estimate the weight of a node would be to use the sum of the weights of all the arcs into a node as the weight estimate. This is termed the flow into the node. Another approach would be to perform a similar procedure for all arcs that leave a node, termed the flow out of the node. For non-sampled profiling, the two estimates should always match (the flow into a node should equal the flow out). However, because of the effects of the sampling process, the two may not be identical.

As with the one- and two-level arc estimates, several schemes for reconstructing node weights were examined. Iterative approaches to a related problem were discussed by Wagner, *et al.* [22]. These approaches propagate node weights through the graph using arc weight information. For example, consider a WCFG region with three nodes: A , B , and C , where A has two outgoing arcs, one to B , and one to C . If A has a weight w_A , and an arc weight of w_{AB} for an arc to node B , and w_{AC} for an arc to node C , then an estimate for the weight of node B would be $\hat{w}_B = (w_A \cdot w_{AB}) / (w_{AB} + w_{AC})$. When loops occur in the WCFG, estimates must be calculated iteratively until they converge. However, reported results suggest that

this iterative inference of node weights from partial information would not produce highly-accurate results [22]. Surprisingly, a relatively simple estimate yielded the highest accuracy: the maximum of the flow in and the flow out as the block weight. One reason for the success of this approach is that, as with arc weights, the relative ordering of blocks by weight is more important to trace selection than the absolute block weight estimate.

3.5 Comparing profiles

No standard metrics exist for the measurement of error in profiles, however one possible method is to examine the effectiveness of hardware-based profiling with a prominent optimization, such as superblock scheduling or trace scheduling. These methods rely on superblock formation and trace selection, respectively, both of which use similar heuristics. However, superblocks differ from traces in the method for providing fix-up code for off-trace/superblock execution and tail duplication [2],[9],[23]. Superblock formation will be considered here, but the techniques presented are applicable to trace selection as well. (An example of superblock formation is resented in Figure 5.)

Superblock scheduling is the process of: (1) superblock formation from a WCFG, and (2) scheduling of instructions inside each superblock. The latter scheduling phase is performed using a variant of list scheduling or other local scheduling techniques [24]. Since scheduling is performed in the scope of a superblock, the nodes that make up the block contribute to the the parallelism or the *instruction per cycle* (IPC) of the code when executed. Because superblock membership is affected by profile information, the IPC metric may be used to

measure hardware-based profiling accuracy. However, error due to hardware-based profiling may or may not influence the ultimate optimization decisions. An absolute metric of error, such as total relative error of all arc or node weights, could be misleading since it may over-estimate or under-estimate the impact of error. Therefore, the experiments that follow use the profiles obtained from hardware-based profiling to drive superblock scheduling. The resultant IPC is then compared with that obtained with perfect (i.e., errorless) profile information. For comparison reasons, basic-block only scheduling is also performed.

Another method used for comparison in the next section is the distribution of arc weight error versus block weights. Although less useful for the reasons discussed above, this metric shows where in the WCFG the trace selection error occurs. It is used as a tool to explain the superblock scheduling results. 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,

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

the (unnormalized) distribution function is,

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

or the sum of the maximum arc differences for each block with weight W .

4 Experimental results

The benchmarks listed in Table III 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 having a 1024-entry buffer (termed *one-level*), and a two-level scheme having a 1024-entry history buffer containing 10-bit history registers (termed *two-level*).

4.1 Sampling error

Error (as defined in the previous section) is due to the characteristics of a statistical sampling experiment. Specifically, hardware-based profiling is an application of cluster sampling, where error is due to two effects: sampling and nonsampling bias [25]. Sampling bias is the component of the difference between the arc weights of the sampled and full profiled distributions due to the sampling procedure. In the context of hardware-based profiling, it is a function of the number of branches between sampling points, the randomness of the sampling points, and the number of sampling points in the experiment.

Nonsampling bias occurs when the population being sampled differs from the actual target population. The actual target population for this study is the traversal of every arc in the WCFG, and the sampled population differs from this due to the methods of arc weight estimation discussed in Sections 3.2 and 3.3, and due to buffer contentions caused by a limited buffer size (1024 entries for this study).

Table IV presents the distribution of arc weight error versus actual (full-profiling) node weights (this is the error metric introduced at the end of the previous section). Arc weights have been separated into two node-weight categories: weights $< 10^4$ and weight $\geq 10^4$ dynamic executions. In addition to the *one-level* and *two-level* results, a third scheme (*counter*) is also shown. *Counter* is an idealized hardware-based profiling scheme in which each entry in the *one-level* branch prediction buffer is replaced with two unlimited-range counters. One counter is incremented every time a particular branch is encountered, while the other counter is incremented every time the branch is taken. In essence, the counters serve as a true measure of arc weights. Since the results from the *counter* scheme are not estimates, the nonsampling bias for this scheme is due only to the limited size of the buffer (1024 entries, as in *one-level* and *two-level*).

As seen in Table IV, the *counter* scheme performs remarkably well for both lightly-executed nodes ($< 10^4$) and high-executed nodes ($\geq 10^4$). Lightly-executed node error is large, indicating branches that are infrequently executed suffer from the limited buffer size. The error in lightly-executed nodes is not a problem in practice since these nodes constitute a small fraction of a program's execution. Highly-executed node error is extremely small, which

indicates that the error introduced due to limited buffer size is not significant, confirming the buffer choice made in Section 2.

For some benchmarks (e.g., *compress*, *cmp*, *eqn* and *qsort*), the two realizable schemes (*one-level* and *two-level*) perform on-par with the ideal *counter* scheme. This result implies that the arc weight estimates of Sections 3.2 and 3.3 perform well. For the two-level scheme, the additional error when compared to *counter* is due to the limited information available in the history registers at each sampling point. Another cause of error is sampling bias, since accuracy also depends on the location of the sampling point in time. As a whole for the two-level scheme, the error is very small for the highly executed nodes and not much greater than the *counter* scheme for the lightly-executed nodes. The *counter* scheme’s low error implies that sampling at kernel entries and context switch intervals is sufficient and does not lead to excessive bias.

All benchmarks that have significant error for the *counter* scheme also have high error for the *one-level* scheme. The relatively higher inaccuracy of the *one-level* scheme is also caused by the additional error due to the heuristic used to record arc weights. The estimate for this scheme cannot access the actual branch history (unlike the *two-level* scheme), which introduces additional nonsampling bias (since the actual population cannot be sampled).

In summary, the component of sampling error due to sample design (sampling bias) is not large, as measured by the experiments above. The remainder of the error is due to *nonsampling bias*, the majority of which is a result of the arc weight estimation methods. Of these, the *two-level* scheme provides higher-quality estimates than the *one-level* scheme.

It should be noted that the error in arc weights does not predict whether a code improving transformation will suffer due to the error—code adjustment across a less-likely path may result in higher overall performance, as is the case in several experiments below.

4.2 Superblock scheduling using hardware-based profiling

Parallelism results for the superblock scheduling experiments are shown in Figures 7 and 8, expressed as instructions per cycle (IPC). In the experiments, the *one-level* and *two-level* hardware-based profiling schemes were used to obtain a WCFG for each benchmark, which was then used to perform superblock scheduling (using the IMPACT architectural framework [26]). The performance results for traditional (errorless) profiling (*full profiling*) are presented, along with the results for scheduling on a per-basic-block basis (*basic-block only*). Figure 7 presents the results for code scheduled for an 8 instruction issue machine having two integer units and one each of load, store, and branch units. Figure 8 presents the results for a 16 instruction issue machine having double the resources of the 8-issue machine. All units have one-cycle latencies except for the load unit, which has a two-cycle latency. For the purposes of relative comparisons, both machines are assumed to have perfect caches (i.e., hit latency = miss latency).

The performance that profiling yields is demonstrated by the difference in IPC between *full profiling* and *basic-block only* which also demonstrates the advantages of profiling itself. The results for *full profiling* improve for a higher issue machine which allows more operations in parallel (Figure 8). In general, hardware-based profiling schemes are able to take advantage

of the additional parallelism in the 16-issue machine, but the schemes do not perform as well as *full profiling*. However, in some cases the hardware-based schemes almost equal the performance of *full profiling* (e.g., *cmp*). The performance for hardware-based superblock scheduling is significantly better than *basic-block only* scheduling for most benchmarks. This demonstrates the performance gain possible using branch handling hardware to collect profile information.

For many of the benchmark results, performance of the *one-level* scheme is almost equivalent to the *two-level* scheme. For some benchmarks such as *compress* and *cccp*, results for *one-level* are better than *two-level*, even though the later branch handling hardware more accurately predicts dynamic branch behavior. To explain this phenomenon, the scheduled code was examined by hand. Figure 9 shows an example from the *cccp* benchmark where the higher error due to *one-level* profiling brought the arc weight from 0.54 (the actual value) to 0.85 (the estimated value) for the node 5 to node 15 transition (part (a) of Figure 9). Since the trace selection threshold is 0.60, the superblock was extended from node 5 to node 15 for *one-level*, but was not extended for *two-level*. The scheduler then had a larger scope in which to perform work (i.e., a superblock consisting of nodes 4, 5 and 15 vs. superblock of nodes 4 and 5). The net result was better performance for the *one-level* scheme.

Another effect that helps explain the phenomenon occurs when the difference between arc counts is small. For example, in several cases the error changed the trace 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 much higher opportunities for code

motion along the less-likely path, resulting in higher overall performance. This fact points more to a non-optimality in scheduling than a shortcoming of hardware-based profiling.

5 Concluding Remarks

This paper has presented hardware-based profiling technique as a method to obtain profile information which does not significantly impact run-time (e.g., slowdown of 0.4%–4.6%). This makes the *compile-use-recompile* approach (presented here) much easier for software vendors than the traditional method of software-based profiling. Using hardware-based profiling, software vendors can supply profiled versions of applications to alpha- and beta-testers, and later collect the profiles and perform final profiled optimizations. No sample suite of inputs is required, since the longer the profiled version remains in the field, the higher the probability that the profiles match actual day-to-day use. Without the need for input sets and the problems of program slowdown, profiling can be used to optimize interactive (e.g., on-line transaction processing), real-time, and system software packages.

Many of the features required to support hardware-based profiling are already present in commercial processors. Most of these processors have branch target buffers, for example, many that employ two-bit counter predictors. The presented results have shown that the error for these schemes is small for highly-executed blocks, which correspond to the blocks that are critical for optimization. The effect of this error is to reduce the performance of superblock scheduling, but a significant performance advantage still results when compared with basic-block only scheduling. It is important to note that the experimental results

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

In general, the techniques presented in this paper significantly reduce the inconvenience of profiling; with contemporary microarchitectures driven increasingly by compiler support, hardware-based profiling will be an important tool for continued improvements in processor performance.

Acknowledgements

This research has been supported by donations from AT&T Global Information Solutions, Intel Corporation, and IBM.

We would like to express our gratitude to the University of Illinois IMPACT group for use of the IMPACT compiler.

References

- [1] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.
- [2] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software–Practice and Experience*, vol. 21, pp. 1301–1321, Dec. 1991.
- [3] W. Y. Chen, *Data preload for superscalar and VLIW processors*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, IL, 1993.
- [4] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling C programs," in *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, (Portland, OR), June 1989.
- [5] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Ann. International Symposium Computer Architecture*, (Jerusalem, Israel), pp. 242–251, May 1989.
- [6] 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] 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.
- [8] 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.
- [9] R. E. Hank, S. A. Mahlke, J. C. Gyllenhaal, R. Bringmann, and W. W. Hwu, "Superblock formation using static program analysis," in *Proc. 26th Ann. Int'l. Symp. on Microarchitecture*, (Austin, TX), pp. 247–255, Dec. 1993.
- [10] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.
- [11] S. P. Song and M. Denman, "The PowerPC 604 RISC microprocessor," tech. rep., Somerset Design Center, Austin, TX, Apr. 1994.
- [12] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, pp. 135–148, June 1981.

- [13] B. Calder and D. Grunwald, “Fast & accurate instruction fetch and branch prediction,” in *Proc. 21st Ann. International Symposium on Computer Architecture*, pp. 2–11, Apr. 1994.
- [14] T. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction,” in *Proc. 24th Ann. International Symposium on Microarchitecture*, (Albuquerque, NM), pp. 51–61, Nov. 1991.
- [15] T. Yeh and Y. N. Patt, “A comparison of dynamic branch predictors that use two levels of branch history,” in *Proc. 20th Ann. International Symposium Computer Architecture*, (Ann Arbor, Michigan), pp. 257–266, May 1993.
- [16] M. L. Golden, “Issues in trace collection through program instrumentation,” Master’s thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 1991.
- [17] M. Smith, “Tracing with pixie,” Tech. Rep. CSL-TR-91-497, Center for Integrated Systems, Stanford University, Nov. 1991.
- [18] J. Larus and T. Ball, “Rewriting executable files to measure program behavior,” *Software Practice & Experience*, vol. 24, pp. 197–218, Feb. 1994.
- [19] T. Ball and J. R. Larus, “Optimally profiling and tracing programs,” Tech. Rep. 1031, Computer Sciences Dept., University of Wisconsin-Madison, 1991.
- [20] 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.
- [21] 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.
- [22] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, “Accurate static estimators for program optimization,” in *Proc. 6th Int’l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, (Orlando, FL), pp. 85–95, June 1994.
- [23] 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.
- [24] 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.

- [25] G. T. Henry, *Practical sampling*. Newbury Park, CA: Sage Publications, 1990.
- [26] 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*, (Toronto, Canada), pp. 266–275, May 1991.

Figure Captions

Figure 1: One-level branch prediction.

Figure 2: SPECint92 benchmarks: Contentions in *BTB* vs. *BTB* size.

Figure 3: Two-level branch prediction.

Figure 4: Example weighted control flow graph. This is an example of a loop that is executed 10 times, with a conditional statement in the body of the loop. Both arc weights and node weights are shown.

Figure 5: Superblock formation for the example WCFG of Figure 4. (The superblocks are shown with a dashed box.)

Figure 6: 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.

Figure 7: Effectiveness of profiling methods for the 8-issue machine.

Figure 8: Effectiveness of profiling methods for the 16-issue machine.

Figure 9: An example from *cccp* showing the error in arc weights for (a) *one-level*, and (b) *two-level* hardware-based profiling. Actual weights are shown in parentheses.

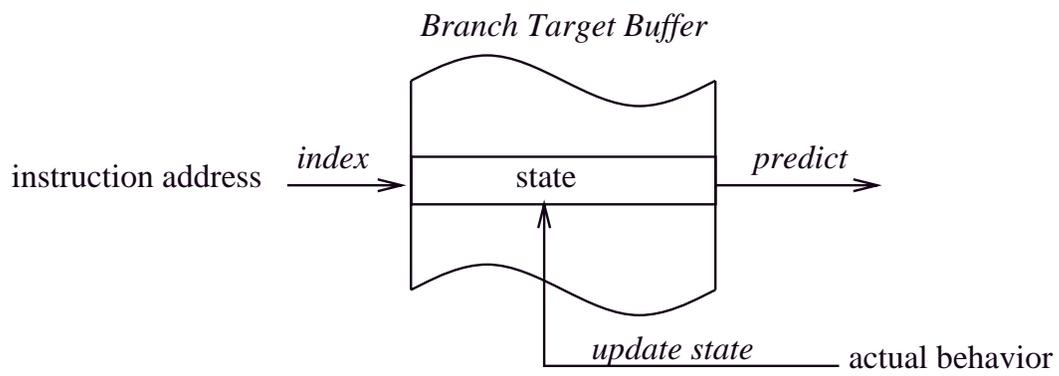


Figure 1: One-level branch prediction.

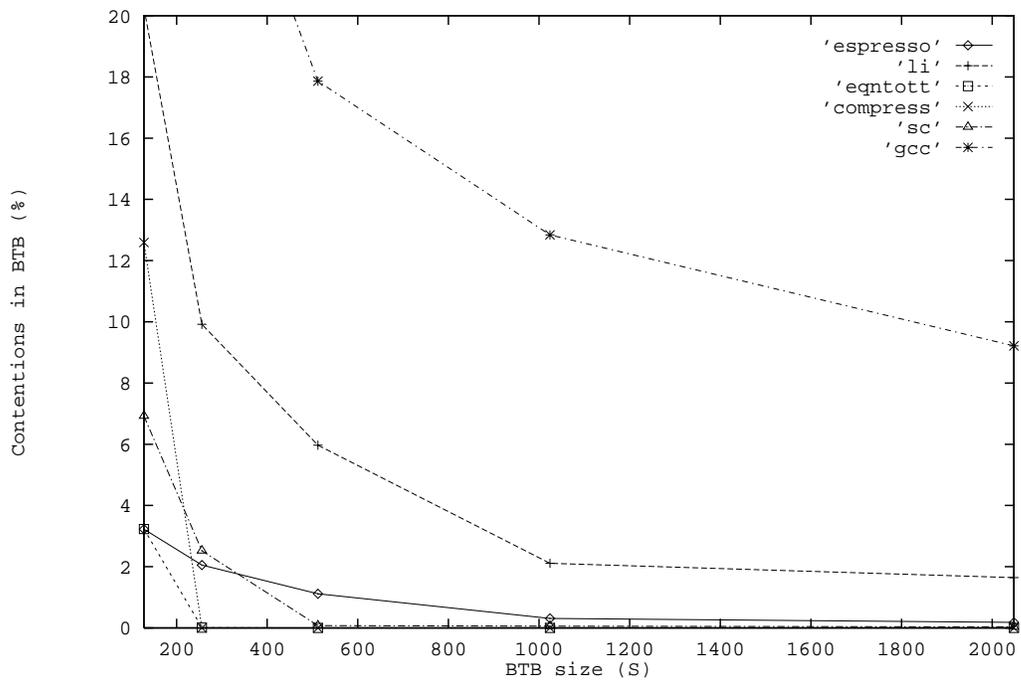


Figure 2: SPECint92 benchmarks: Contentions in *BTB* vs. *BTB* size.

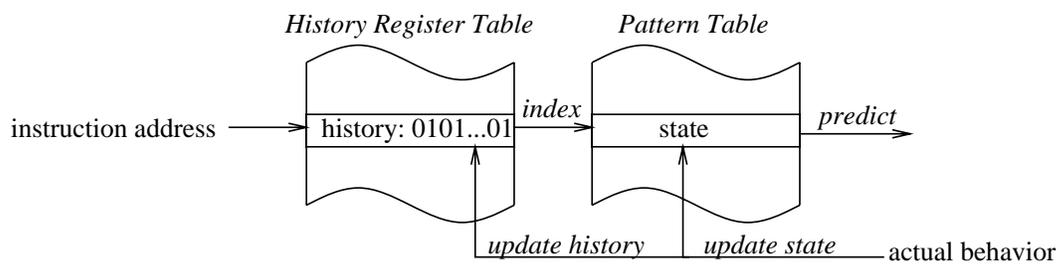


Figure 3: Two-level branch prediction.

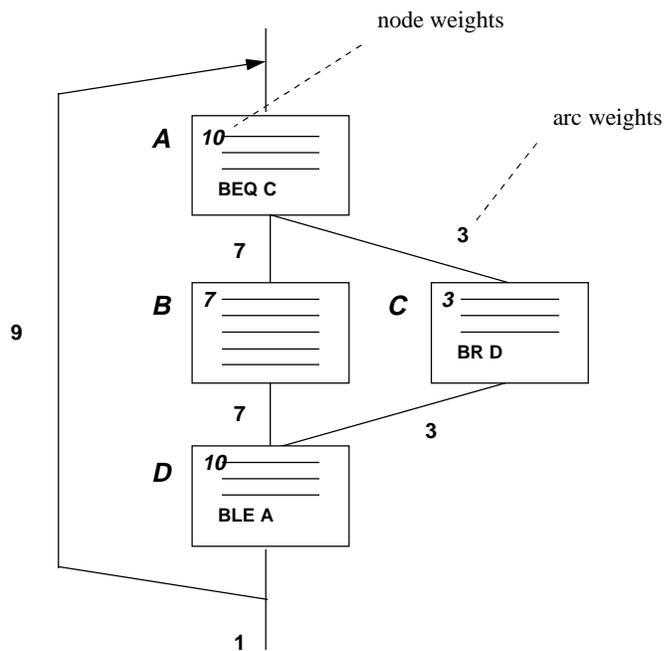


Figure 4: Example weighted control flow graph. This is an example of a loop that is executed 10 times, with a conditional statement in the body of the loop. Both arc weights and node weights are shown.

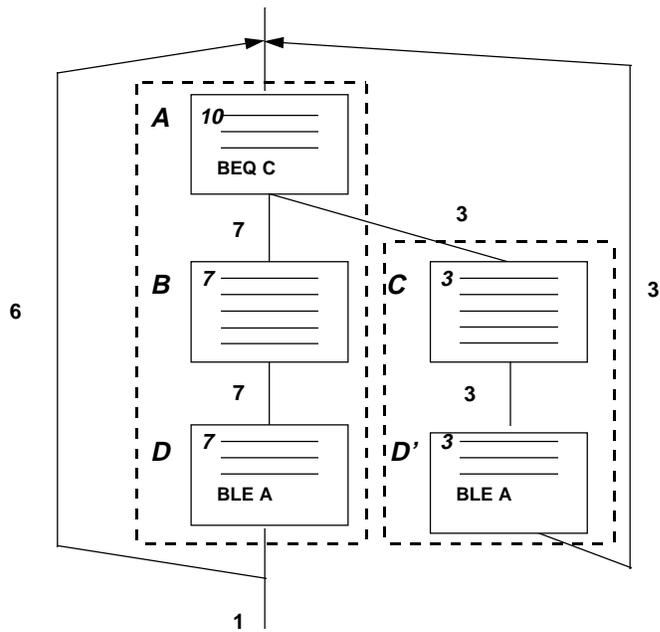


Figure 5: Superblock formation for the example WCFG of Figure 4. (The superblocks are shown with a dashed box.)

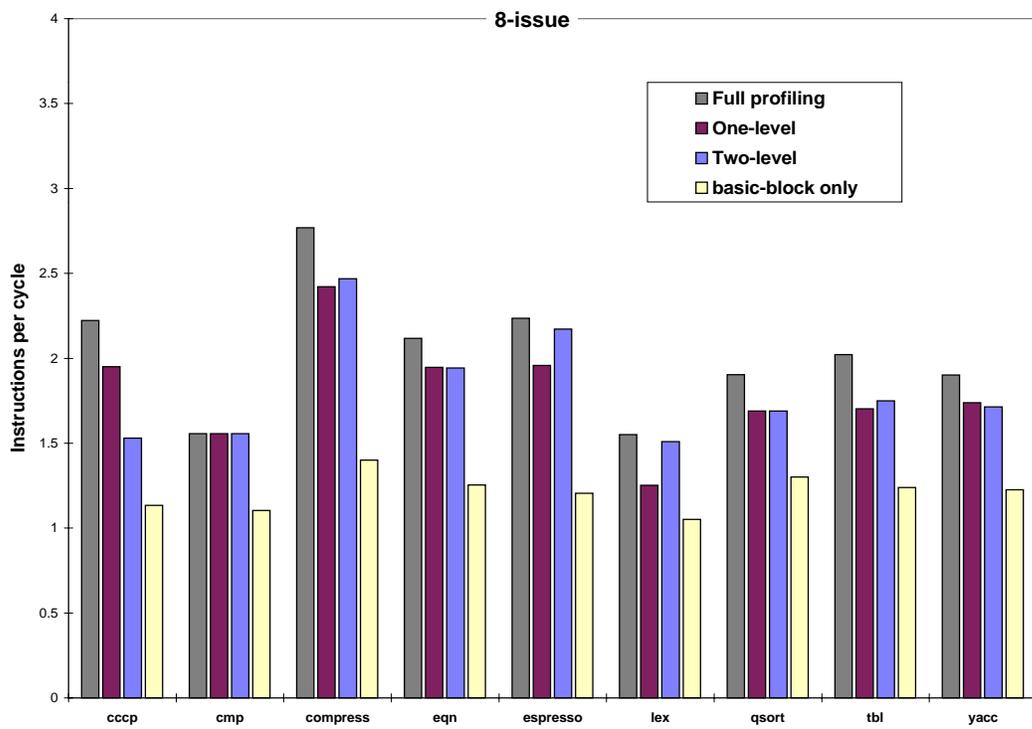


Figure 7: Effectiveness of profiling methods for the 8-issue machine.

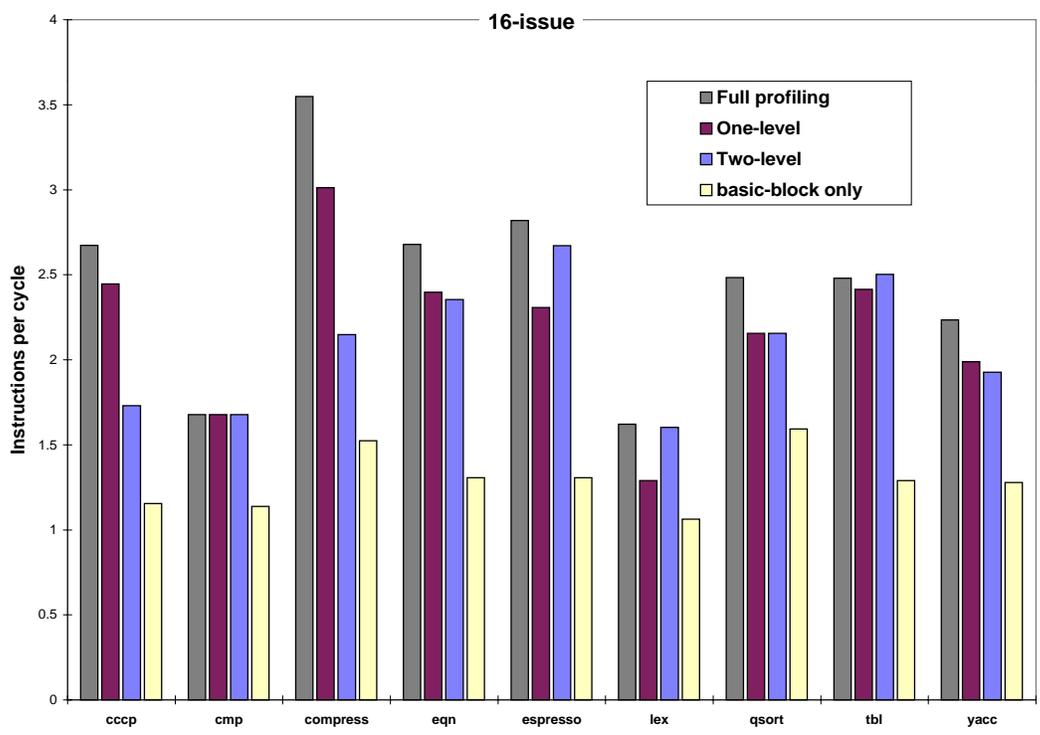


Figure 8: Effectiveness of profiling methods for the 16-issue machine.

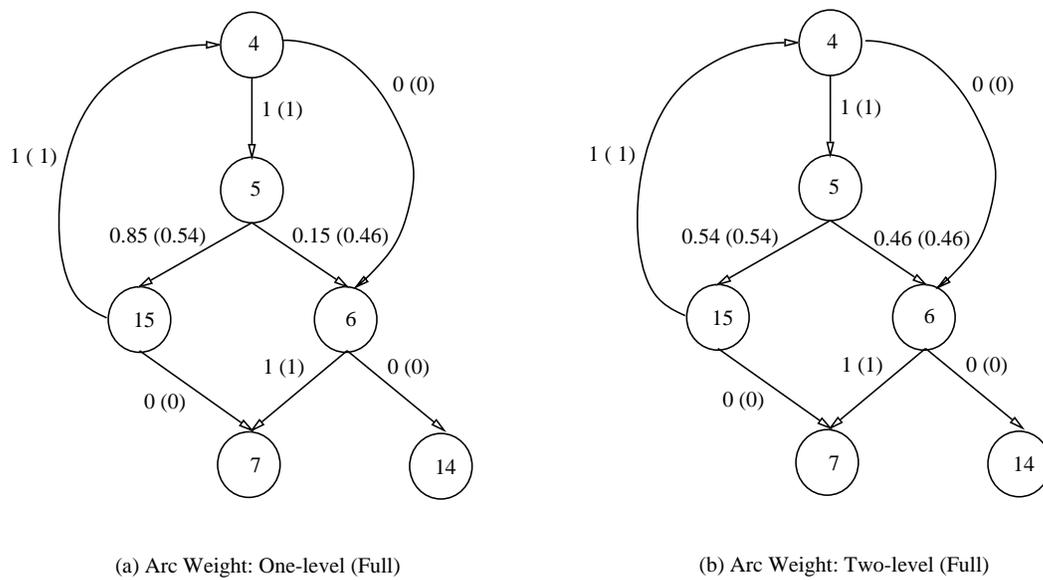


Figure 9: An example from *cccp* showing the error in arc weights for (a) *one-level*, and (b) *two-level* hardware-based profiling. Actual weights are shown in parentheses.

Table I: Results of 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%

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

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

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

Table III: Benchmarks used for evaluation of profiling methods.

Benchmark	Description
cccp	GNU C-compatible compiler preprocessor
cmp	file comparison
compress	file compression utility
eqn	mathematics typesetting
espresso	minimization of boolean functions
lex	lexical analysis program generator
qsort	sorting utility
tbl	table formatting for nroff
yacc	parser generator

Table IV: Error in arc weight estimation. (Columns marked “ $< 10^4$ ” account for error in blocks with actual weight less than 10^4 number of dynamic executions, whereas columns marked “ $\geq 10^4$ ” denote error in blocks with greater than or equal to 10^4 executions.)

Benchmark	One-level		Two-level		Counter	
	$< 10^4$	$\geq 10^4$	$< 10^4$	$\geq 10^4$	$< 10^4$	$\geq 10^4$
cccp	25.81	0.59	4.59	0.20	2.64	0.00
cmp	2.0	0.03	2.0	0.01	2.0	0.00
compress	8.06	2.17	6.08	1.23	6.0	0.0
eqn	10.82	1.13	10.26	0.21	8.33	0.00
espresso	88.69	55.81	36.55	22.63	9.36	0.49
lex	42.66	12.02	26.88	2.86	13.10	0.01
qsort	1.0	2.02	1.0	1.19	1.0	0.0
tbl	74.76	0.47	45.47	0.01	24.90	0.0
yacc	49.21	24.15	26.40	8.13	5.45	0.08