Commercializing Profile-Driven Optimization

J. Stan Cox* David P. Howell* Thomas M. Conte†

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

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

Abstract

There are a broad selection of code-improving optimizations and scheduling techniques based on profile information. Industry has been slow to productize these because traditional ways of profiling are cumbersome. Profiling slows down the execution of a program by factors of 2 to 30 times. Software vendors must compile, profile, and then re-compile their products. In addition, profiling requires a representative set of inputs and is hard to validate. Finally, profiling has had little success for system code such as kernel and I/O drivers.

This paper discusses experiences AT&T Global Information Solutions has had with commercializing profile-driven optimizations. Three approaches to profiling are discussed, along with results and comments concerning their advantages and drawbacks. The validity of profiling is discussed. One new innovation, hardware-based profiling, removes many of the problems vendors have with profiling. The paper also discusses methods to profile system code and support debugging. In general, the data and techniques presented in this paper can be used to productize profiling and advocate its use to the software business community.

1 Introduction

Advanced compilers perform optimizations across block boundaries to increase instruction level parallelism, enhance resource usage and improve cache performance. Many of these methods, such as trace scheduling [1],[2] superblock scheduling [3] and software pipelining [4] 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 [3]. Other examples include data preloading [5], improved function in-lining [6], and improved instruction cache performance [8].

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

Execution of the profiled version of the program is slow. With some methods, the profiled version runs 30 times slower than the optimized program. In addition it is difficult to choose and validate the test inputs used for profiling [9],[10]. Even given the obvious performance advantages, software vendors are hesitant to use profile-driven optimizations due to the added complexity of development, maintenance, support, and debugging.

Static estimation solves some of the problems related to gathering profile data [11]. However, these techniques are not as accurate as profiling [9],[10]. When used for superblock scheduling, static estimates achieve approximately 50% of the speedup that profiling can achieve [7]. In addition to this, static estimation cannot capture highly data-dependent branches, limiting the acceptance of this approach by software vendors.

AT&T Global Information Solutions is a server system company. Our key applications are third-party database products. Many of the published solutions to profiling's problems do not apply to our business. This paper discusses the how our profiling techniques evolved based on our experiences with third-party vendors. Several profiling techniques are presented and compared, along with our approach to commercializing profile-driven optimization.

2 Profiling Techniques

Techniques for implementing profiling often trade accuracy for profiled code speed. We started with highly accurate techniques, explored less-accurate methods, and gained experience in the advantages and disadvantages of each approach.

2.1 Probe-based profiling

Our first attempt at basic block profiling used the technique of gathering a trace of basic block id tokens, and then interpreting the trace and counting the transitions made between the basic blocks. In this technique, the transition data is written out to a data file organized by modules, functions in each module, and basic blocks in functions with transitions to other basic blocks and corresponding counts. The trace contains *handle* tokens and basic block tokens. To fully qualify a basic block requires a *handle* that holds string pointers to the module name and function name for the block, followed by a block number token. The handles and block tokens both use 32 bit containers. To differentiate them, the basic block numbers use the bottom 16 bits with the upper set to zero, and handles use the full 32 bits with the upper 16 always non-zero.

To shorten the trace, a handle is only generated if a transition out of the function is made, otherwise a stream of basic block number tokens are generated. For example, consider Figure 1. If the execution transitions from blocks **3** to **4**, calls function `bar()`, returns to **4**, and then goes onto **9**, this results in the trace: **ID1, 3, 4, ID2,** ..., **ID1, 4, 9**. This is then converted on-the-fly into a basic block transition count tree structure, shown in Figure 2. The profile data file follows the same structure as the transition count tree, threaded with module pointers.

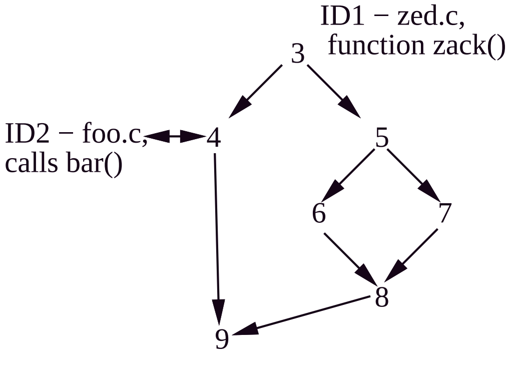


Figure 1: Probe-based profiling example.

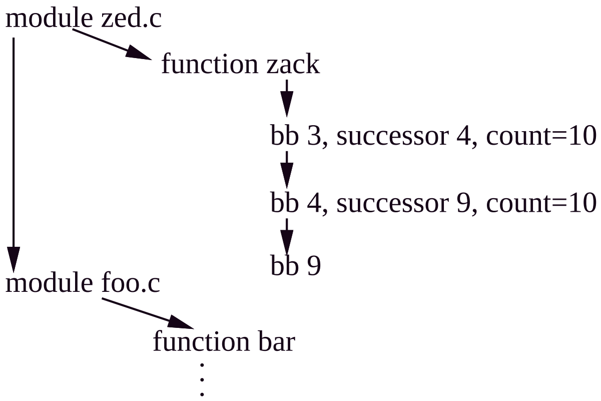


Figure 2: Transition count tree structure.

We designed the profile-driven optimizations around basic block transition count statistics since we felt that this was an accurate method for determining the behavior of an application for later optimization. A design goal of the profiling scheme was to provide flexibility of information collection and processing. To that end, either a trace or the summerized basic block transition count can be generated by switching the profile data handling library and relinking. Due to the size of the unprocessed trace, the transition count form of the profile data is used for the optimizer.

The optimizer reads in the profile data after building the flow graph for the program and correlates it with each node and node successor in the flow graph. Optimizations are then driven using the weighted basic blocks.

The primary weakness of this technique is the run time overhead incurred for profiling (about 26× on average, measured using SPECint92 benchmarks– see Section 2.5 below for the full data). Another major weakness is the inaccurate conversion from the trace to transition counts due to asynchronous events. These events include signal handlers, interrupts, and thread managers. This especially limits the optimizer's use of the profile data and forces the invalidation of the profile data for functions where it occurs. Many of our applications have high amounts of I/O, and this presented a major problem. This profiling scheme is also unacceptable for several key applications where asynchronous flow changes are common, like the Unix kernel and applications that use threads packages. We did manage to gather good profile data for these applications, but not without significant modifications to the profiling and application code.

Another problem is skewed program results due to the overhead of performing profiling. The few applications we tried to optimize with profile data didn't realize the full potential of profile-driven optimization due to timing impacts of profiling overhead.

We were able to successfully profile major commercial database products, with no change to the profiling implementation or application. The results of optimization using this data was very promising. This supplied the motivation to develop other profiling schemes with less shortcomings.

2.2 Node-based profiling

Node-based profiling is used in other commercial compilers that we have evaluated. In this technique, every basic block entrance is recorded by incrementing a counter, with one unique counter per block. A database that represents the program's control flow graph is generated by the compiler. Once the profiled program is run, the database with counts is read in by the optimizer to weigh each node in the flow graph and optimize accordingly.

While quick and effective, the node-based technique has some problems. The most-notable problem is the lack of block-to-block transition information. Counting the basic block transitions would give us a more accurate representation of the behavior of the program and weighting of the flow graph for optimization. Although transition counts can be approximated from the block counts, we found that more accuracy was required.

2.3 Arc-based profiling

The arc-based profiling approach counts the *dynamic transitions* between basic blocks during execution. This is similar to the node based approach except that the arc transitions are counted, instead of the block entrances themselves. To achieve this profiling technique, the compiler places probe instructions into the program that count each block transition. As in the node based approach the probe is simply a sequence of instructions which increment the corresponding element in the static table. For each basic block, if it has only one successor block, then the probe can be placed in the block. If the block has multiple successors, then the probe can be placed in a successor if the successor's only predecessor is the block. For each case where the block has multiple successors and a successor has multiple predecessors, a new block has to built and placed between the block and the given successor.

Table 1: Static control flow graph statistics for SPECint92 and arc-based profiling.

Benchmark	blocks	transitions	addional trans. blocks
compress	356	511 (144%)	118 (+33%)
eqntott	1032	1529 (148%)	482 (+47%)
espresso	6134	8791 (143%)	2803 (+46%)
gcc	22282	34633 (155%)	13199 (+59%)
li	2106	2819 (134%)	1026 (+49%)
sc	3029	4652 (154%)	1375 (+45%)
avg.	5823	8823 (152%)	3167 (+54%)

Table 1 shows the static number of basic blocks, block-to-block transitions, and additional transition blocks required for arc-based profiling for the SPECint92 benchmarks. Surprisingly, the average number of transitions is only approximately 1.5 times the number of blocks. Furthermore, the number of transition blocks that have to be added for this profiling approach is not excessive (an increase of 54% in the number of blocks). This makes the arc-based scheme an efficient and accurate method to perform basic block profiling. Unlike the node counting approach, the arc based approach is completely accurate. This accuracy is obtained by requiring a slightly larger static table. In addition, arc-based profiling has a slightly larger increase in run-time slowdown over node-based profiling. One minus is some post-processing is required to extract node weights from the arc-based profile information.

2.4 Hardware-based profiling

One significant revelation for us was that the entire information needed for the arc-based approach was already maintained inside the branch-target-buffer (BTB) of our processor. The BTB is indexed by the address of each branch and stores the most-recent target address of the branch. Thus, each entry in the BTB comprises a (source address, destination address)-pair, which is easily translated to identify specific arcs. In addition to this, the prediction information maintained is an approximate count of the number of times the arc was executed, although some post-processing is required.

Hardware-based profiling can be used with existing processors by exploiting BIST scan paths or performance monitoring features. The accuracy of the scheme is not as high as probe-based or arc-based profiling, since the BTB must be sampled. This is discussed in depth in [15]. One reason for the error is that short-lived arcs are not well represented in the sampled data. However, they comprise the less-frequently executed transitions in the program. The error is quantified in the following section.

2.5 Comparisons

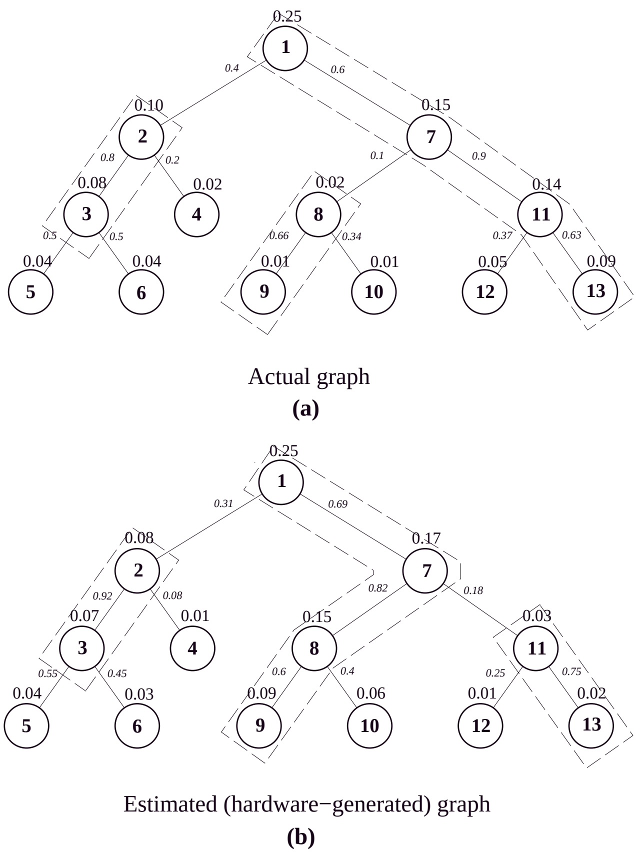
The slowdown of each profiling technique for SPECint92 is shown in Table 2. Clearly, probe-based profiling is the worst, exceeding 37× for *espresso*. Node-based and arc-based profiling are relatively close, with arc-based profiling nearly as fast. Hardware-based profiling is by far the fastest, with two benchmarks showing no appreciable slowdown whatsoever.

Table 2: Slowdown of the profiling techniques.

Benchmark	probe	node	arc	hw
compress	14.2	1.36	1.71	1.03
espresso	37.2	1.51	1.56	1.05
eqntott	28.4	1.71	1.79	1.01
gcc	25.4	1.79	1.83	1.03
li	33.1	1.21	1.27	1.00
sc	19.6	1.31	1.32	1.00

The accuracy of the four techniques differ. In the absence of asynchronous events, probe-based profiling and arc-based profiling are perfectly accurate. In the presence of such events, arc-based profiling has superior accuracy. Node-based profiling is much less accurate than either probe- or arc-based because it does not fully model transitions between blocks. Hardware-based profiling is inaccurate, but for a different reason: it is forced to sample the execution instead of processing a full trace of all transitions. Nevertheless, the near-zero slowdown of the hardware technique allows it to be used in ways that the other techniques cannot. For example, it can be used without the users' knowledge– allowing an application to be installed for some period of time then later retrieved. Once retrieved, its accumulated profile data can be used to re-optimize the code based on actual usage.

Validation of hardware profiling is done by comparing traditionally-generated profiles (*actual profiles*) to hardware-generated profiles (*estimated profiles*). One method for this is to perform *trace selection* on both the actual and the estimated profiles and compare the results. An example of trace selection is illustrated in Figure 3. Graph (a) is annotated with the actual profile information, whereas graph (b) is the hardware-generated profile. Traces are formed using an arc trace selection threshold of 60% to group blocks [14]. Code explosion is avoided by not extending traces to blocks with low weights. This is also implemented as a threshold. Threshold values of 0.1%, 1%, 3% and 5% are considered below. The lower this threshold, the larger the code size of the generated executable, since additional patch-up code is required when instructions are moved inside a trace.



Actual graph

(a)

Estimated (hardware−generated) graph

(b)

Figure 3: Trace selection example.

The metric for trace selection error is introduced using the example of Figure 3. In the actual graph (graph (a)), basic blocks **1**, **7**, **11** and **13** are grouped together to form a trace. Due to errors in the weights of outgoing arcs for block **7**, the blocks **1**, **7**, **8** and **9** are grouped to form a trace in the estimated graph. The error for block **7** is due to the difference in arc weights between the two graphs. The transition from block **7** to **8** will occur 0.15 × 0.1 = 1.5% of the total execution time. Similarly, the transition from **7** to **11** will occur 0.15 × 0.9 = 13.5% of the time. (Since the actual graph contains the real execution frequencies of the program, these frequencies are used.) Hence, the transition from **7** to **11** occupies a higher percentage of the total execution. The trace in graph (b) incorrectly assumes the transition of **7** to **8** is more likely. This assumption is wrong for $13.5\% - 1.5\% = 12\%$ of the execution. The figure of 12% is therefore the percentage of execution time that the incorrect trace membership will be exercised. In general, the trace selection error is the total percentage of execution time that incorrect trace membership is exercised due to errors in the estimated profile. Table 3 presents the trace selection error for the SPECint92 benchmarks.

The worst-case error is for *espresso* with a very relaxed cutoff threshold of 0.1%. The 18.49% in Table 3 is the amount of program execution where the trace selection would differ. Examination of the sources for this error revealed an interesting trend. For example, in *compress* a trace composed of blocks **33-36-37-38** in the actual profile was split into two traces between blocks **36** and **37** in the estimated profile. This occurred because of an error in the estimated arc frequency between blocks **36** and **37**. In this case the weight of this arc was less than the trace selection threshold, preventing the trace to grow beyond block **36**. Such errors in trace selection have the effect of reducing the scope of the profile-based optimizations, which has few detrimental effects.

Another method for comparison is the distribution of arc weight error versus block weights. This metric is useful since it shows where the trace selection error is occurring. The distribution is calculated by computing the maximum differences between the actual and the estimated arc weights for each category of block frequencies. 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. The distribution of arc weight error provides good insight into the performance of the hardware technique, as is shown in Figure 4.

Two features are evident in the figure. First, the majority of the error is for low-weight blocks. This appeals to intuition, since infrequently-executed branches will be sampled less frequently by the technique. A second feature is the lack of error between block weights of approximately 0.3 and 1.0. This indicates that very heavily-weighted blocks are captured correctly. Even the magnitude of the error for moderately small weights (e.g., 0.001) is small and therefore non-critical ($< 15\%$).

3 Commercialization Issues

The performance advantage obtained with profile driven optimizations are dramatic and have been presented in several papers [1],[2],[3],[4],[5],[6],[8]. Software vendors and developers of complex applications are, however, hesitant to use profile-driven optimizations. Their concerns vary, but usually involve the worry that the optimizations may not improve the performance of the application for all input sets, or could actually slow their application down for differing input sets. Developers are also uneasy about what input sets to choose for profiling. Current research suggests however that these concerns not critical. In particular, profile-driven optimizations improve program performance for all inputs.

Hwu, *et al.* [12] reported that with profiling they were able to obtain branch prediction accuracy comparable with much more expensive hardware mechanisms. They also report that 98% of the advantage of prediction was preserved across runs with varying input sets.

More research in this area was performed by Fisher and Freudenberger [9] and Wall [10]. Fisher and Freudenberger reported that "*...even code with a complex flow of control, including system utilities and language processors written in C, are dominated by branches which go in one way, and that this direction usually varies little when one changes the data used as the predictor and target.*"

Their experiments compared a run of an optimized application using an input from the profiling input set versus a run using an input not in the profiling set. Their data showed an application using an input not in the profiling set can expect to achieve 75% or more of the benefit obtained from the ideal case. Fisher and Freudenberger reported that branch prediction "*can be done almost as well as is possible by taking previous runs of a program, and using those runs to make decisions about which way branches will go in future runs.*" Similarly, Wall concluded that profiles used to predict program behavior for different run characteristics might not do as well, but was "*often quit close, however and was usually at lease half as good.*" Wall's data backs up Fisher's results of 75% of the benefit of use a "perfect" or identical data set run for runs with differing input.

The examples and conclusions above are further supported by work which is on-going at the University of Illinois. In [3], the different inputs were used to profile from the ones used to measure performance. Good performance benefits were sustained across these inputs. In [13], the same type of experiments were performed for superscalar processors. Again, profile-driven optimizations with different inputs from the profiling run sustained good results.

If the hardware-based profiling approach is used, the problem of input set selection is removed. This

Table 3: Hardware profiling - trace selection error (percent).

Benchmark	Trace selection error			
	Threshold: 0.1%	Threshold: 1%	Threshold: 3%	Threshold: 5%
espresso	18.49	5.51	0.02	0
xlisp	8.51	2.30	0	0
eqntott	3.66	3.64	3.64	0
compress	11.92	10.89	2.65	2.65
sc	3.46	1.14	0	0
gcc	5.99	0	0	0

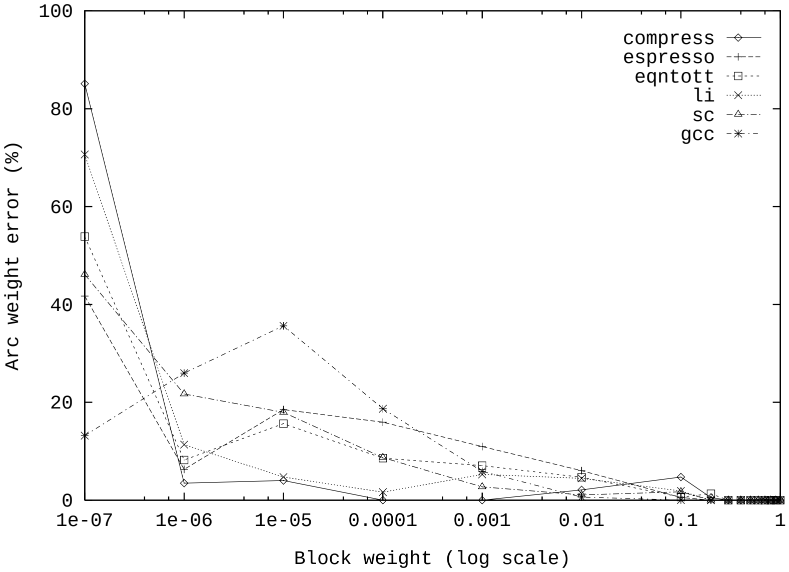


Figure 4: Distribution of arc weight error for hardware profiling.

is a direct consequence of the speed of this approach, which has a nearly imperceptible slowdown. Using hardware-based profiling, the application can be beta tested in a "real-world" environment and thus is exercised as it would be once released. The collected profile information would accurately capture the actual usage characteristics of the application. This approach is also more effective for real-time and other time-based applications.

3.1 Support

Another point of concern which prevents vendors from using profile-driven optimizations is the added effort for support. Gathering profile data to use for optimization is a time consuming process but is easily justifiable given the benefits obtained. Unlike the SPEC92 benchmarks, real applications' source code changes over time due to bug fixes and feature enhancements. These changes can in turn invalidate profile data that has been gathered for the application.

Software vendors are justifiably unwilling to reprofile an entire application for each bug fix and code change over the lifetime of an application release. On the other hand, software vendors cannot allow the performance of an application to decline over time because of code changes which invalidate profile data.

To alleviate this problem, it is important that the compiler invalidate profile data at the function or block level. If the profile data used is based on source lines and module names in the application, then a single bug fix can invalidate the profile data for an entire module. This is clearly unacceptable. The profile data should not be based on source line numbers and should invalidate only those functions that have changed due to a code change. Furthermore, it is easy for the compiler, based on the given profile data, to warn the developer when a frequently executed or "hot" function has been invalidated due to a source change. This profile set is then used by the compiler to optimize over the lifetime of the release. If code changes are made, only the functions involved will have their profile data invalidated. The compiler will warn when a "hot" functions profile data is no longer valid, and only in this case will the process of profiling be repeated.

3.2 Debugging

Debugging is also a major concern for software vendors. Relating assembly code back to source code seems to be a major concern. This is hard to do in the presence of the block-reordering optimizations performed by our compiler.

To address these issues and to enable additional tools to be written to utilize the basic block data, we have implemented a basic block table that describes each basic block in an application. Profiling is done using references to a basic block transition table, where each table slot is a (from, to) pair of indices to basic block entries that represent a transition between two basic blocks. A simple counter for each transition represents the profile data counts for the transition.

The basic block table entries carry the basic block handle which gives the module name and function name, the block number, the start text address, its size, and the start source line number. An entry is created each time that the compiler builds a basic block. Likewise, every time a block ends, a transition table entry is generated to its successors, and probes are inserted. Blocks and transition table entries from modules are appended together by the linker when an executable is built.

This structure enables a number of debugging and profiling tools. For example, an application that has been optimized will have very little correlation back to the source code due to block rearrangement. The basic block table can be used to look up the block number (each table entry has a start address and size) and display the source module, function, and line number where the block starts. This enables source level debugging for block-reorganized code.

Profiling tools can also make use of the basic block and transition to show program hot spots and hot paths. The transition counts can be correlated with the block counts to show hot basic blocks, or to show execution coverage of a set of blocks, which can then be related back to the source using the basic block table. The transition table, which is really a flattened flow graph, can be used to show the hot program traces or to show weighted arcs between blocks with a hypertext-type tool. These tools help developers realize the best performance from their applications and are an added benefit of profiling technology.

4 Concluding Remarks

Profile-driven optimizations have untapped potential for improving commercial application performance. Much of this is due to skepticism from software vendors based primarily on profiling input set selection, profiling performance, and problems with debugging and support.

We have implemented the major profiling approaches at AT&T Global Information Solutions and evaluated their tradeoffs. Several techniques have clear advantages over the others. Where before the overhead and intrusiveness of our original profiling made it impossible to profile the full Unix kernel, the new arc- and hardware-based implementations have made kernel profiling practical. Multithreading issues are also resolved by the new implementation as arc counts can represent any kernel process (or processor).

In terms of slowdown, arc-based profiling is nearly as fast as node-based profiling, yet more accurate. In addition, it handles asynchronous events such as interrupts and thread managers. Hardware-based is best for speed, allowing a new style of profiling where the program is installed and profiled unbeknownst to the users, then later collected and recompiled. This technique has been well-received by practicing database engineers.

5 Acknowledgements

Thanks to Marv Graham, Lorraine Lee and Linda Gray (of AT&T Global Information Solutions), and Burzin Patel (graduate student at South Carolina) for comments, suggestions and support.

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

[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] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proc. 14th Annual Workshop on Microprogramming*, pp. 183–198, Nov. 1981.

[5] 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.

[6] 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.

[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*, (Austin, TX), pp. 247–255, Dec. 1993.

[8] 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.

[9] 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.

[10] 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.

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

[12] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proc. 16th Ann. International Symposium Computer Architecture*, (Jerusalem, Israel), pp. 224–233, May 1989.

[13] W. W. Hwu, *et al.* "The superblock: An effective structure for VLIW and superscalar compilation," *Journal of Supercomputing*, pp. 229–248, July 1993.

[14] 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.

[15] T. M. Conte, B. A. Patel, and J. S. Cox, "Using branch handling hardware to support profile-driven optimization," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, (San Jose, CA), Dec. 1994.