The Effect of Program Optimization on Trace Cache Efficiency

Derek L. Howard and Mikko H. Lipasti
IBM Server Group
Rochester, MN 55901
derekh@us.ibm.com, mhl@ece.cmu.edu[1]

Abstract

Trace cache, an instruction fetch technique that reduces taken branch penalties by storing and fetching program instructions in dynamic execution order, dramatically improves instruction fetch bandwidth. Similarly, program transformations like loop unrolling, procedure inlining, feedback-directed program restructuring, and profile-directed feedback can improve instruction fetch bandwidth by changing the static structure and ordering of a program's basic blocks. We examine the interaction of these compile-time and run-time techniques in the context of a high-quality production compiler that implements such transformations and a cycle-accurate simulation model of a wide issue superscalar processor. Not surprisingly, we find that the relative benefit of adding trace cache declines with increasing optimization level, and vice versa. Furthermore, we find that certain optimizations that improve performance on a processor model without trace cache can actually degrade performance on a processor with trace cache due to increased branch history table interference. Finally, we show that the performance obtained with a trace cache of a given size can be obtained with a trace cache of about half the size by applying aggressive compiler optimization techniques.

1. Introduction

Achieving a high degree of instruction-level parallelism with general-purpose integer programs requires an aggressive, high-bandwidth instruction fetch mechanism that delivers a large number of useful instructions to the processor core in every cycle. Examining how to build such fetch mechanisms has been and continues to be an important and popular area of research (e.g. [1,2,3]). An effective instruction fetch mechanism must solve three problems in order to deliver high instruction bandwidth: it must accurately predict multiple conditional branches in each cycle; it must fetch multiple taken branch targets per cycle; and it must do so within the confines of ever-shrinking cycle time budgets. The latter requirement is obvious, while the first two follow from the fact that general-purpose programs have frequent conditional branches and frequent taken branches. One of the most promising recently-proposed approaches for solving all three of these problems is the trace cache [2,4]. A trace cache stores instruction sequences in dynamic execution order, rather than in static program order, effectively folding out taken branches from the fetched instruction stream. The trace cache fill unit collects dynamic sequences of instructions (*traces*) as they execute, and then allocates space for these traces in the cache. Subsequent fetches access the trace cache with the same fetch address that is sent to the instruction cache, and a directory tag match returns a block of instructions that may contain multiple conditional and/or taken branches. Multiple traces starting at a given fetch address can be stored in the cache by adding a vector of branch direction bits to the tag entry. One of these entries can then be selected by performing multiple branch predictions during the fetch and using those predictions to select the matching trace entry. This approach is termed *path associativity* in the literature.

While trace cache has been shown to be an effective dynamic technique for mitigating the effects of taken branches, static techniques for accomplishing the same goal have existed for years. Compiler techniques such as loop unrolling (e.g. [5]), aggressive procedure inlining (e.g. [6]), and feedback-directed program restructuring and optimization (e.g. [7, 8, 9]) are widely available and known to provide significant performance benefits. Not only do these optimizations increase the compiler's scope or visibility into larger portions of the program, hence opening up additional opportunities and degrees of freedom for traditional code transformations, but they are also very effective at reducing the number of taken branches that a program executes. Hence, one would expect these code transformations to have a significant impact on the behavior and efficacy of dynamic techniques like trace cache. Interestingly enough, we know of no study that examines this problem in detail. It is the intent of this paper to examine this question in the context of a high quality, optimizing, production compiler that is in widespread commercial use--the IBM xlC compiler for AIX version 4--and a cycle-accurate simulation model that implements trace cache.

2. Compiler Optimizations

The set of compiler optimizations studied in this paper include loop unrolling, automatic procedure inlining, profile-directed feedback, and feedback-directed program restructuring. All of the optimizations are applied separately as well as in combination to measure their effects as well as their interactions. Furthermore, they are all applied in the context of the conventional baseline optimizations provided by the xlC compiler. These baseline optimizations include

1. Now with the University of Wisconsin - Madison

TABLE 1. Optimization Effects. Code size and speedup are relative to baseline optimization.

	Inline10		Unroll8		PDF		FDPR		Combined	
Benchmark	**Code Size**	**Speedup**	**Code Size**	**Speedup**	**Code Size**	**Speedup**	**Code Size**	**Speedup**	**Code Size**	**Speedup**
go	107.62%	0.974	110.80%	1.009	101.12%	1.023	101.67%	1.039	124.15%	0.996
m88ksim	102.90%	1.045	101.88%	0.991	101.94%	1.000	100.07%	1.098	107.26%	1.075
gcc	112.41%	0.979	107.12%	0.970	103.86%	0.960	99.23%	1.224	111.65%	1.179
compress	100.75%	1.000	100.46%	1.026	99.89%	1.026	99.95%	1.049	100.24%	1.107
li	109.85%	1.170	101.51%	1.015	100.59%	1.024	99.22%	1.011	111.19%	1.215
ijpeg	105.55%	1.016	101.71%	1.025	100.94%	1.055	99.17%	1.102	112.33%	1.087
perl	103.24%	1.007	104.94%	1.014	103.01%	1.004	100.42%	1.127	112.81%	1.169
vortex	104.13%	1.044	1.0187%	0.959	105.14%	1.044	97.85%	1.091	108.05%	1.080

eliminating local common subexpressions, constant folding, code motion, constant propagation, register allocation, and a host of other optimizations that one would expect to find in an industrial-strength, production-quality workstation compiler. We used runtime experiments on an RS/6000 model 143P workstation to guide our selection of loop unrolling and procedure inlining; the runtime and code size effects of our selections are summarized in Table 1.

Loop unrolling. Loop unrolling involves identifying and replicating static loop bodies multiple times, and adjusting the loop termination code and iteration counts to preserve semantic correctness. For example, loop unrolling by a factor of two statically writes the instructions for two loop iterations, and reduces the number of iterations by two. The baseline optimization will unroll loops up to degree four. We used the compiler's maximum allowable unrolling degree of eight in order to emphasize the effect of loop unrolling in our subsequent trace cache experiments.

Procedure Inlining. Automatic procedure inlining copies the instructions from a called procedure into the calling procedure, inserting them at the call site. Doing so avoids the call overhead and two taken branches, and exposes additional optimization opportunities to the compiler. The xlC compiler allows arbitrarily large procedures to be inlined by specifying a maximum size threshold. The baseline optimization will not automatically inline procedures at all. We used a threshold value of 10 in subsequent microarchitectural experiments to avoid excessive code bloat.

Profile-Directed Feedback Optimization. Profile-directed feedback optimization (PDF) collects profile information on the behavior of the program when run with a representative input set. This profile information is used to improve code near conditional branches and in frequently executed code sections by filling branch delay slots. When PDF is applied, the instruction scheduler exploits knowledge of frequently-occurring paths through the code to fill idle issue slots caused by branch delays with instructions that are hoisted speculatively from the most likely path following a conditional branch. The PDF optimizations are applied only within the scope of a single compilation unit. To avoid clouding our experiments with inaccurate profile information, we use the same input sets for profiling and subsequent experiments.

Feedback-Directed Program Restructuring. FDPR is a separate post-pass code object-code optimization tool that is available under AIX version 4 [9]. FDPR collects profile information on the behavior of the program as it runs and uses it to restructure the program, reorganizing basic blocks to minimize taken branches based on branch profiles and moving unused or infrequently used blocks out of line to prevent them from polluting the instruction cache. FDPR is applied globally, to the whole linked executable as well as statically-linked library code, hence deriving further benefit beyond PDF, which is applied separately to each compilation unit. The code transformations made by FDPR are not influenced by the specific instruction cache organization of the target machine; rather, the transformations attempt to coalesce frequently-executed code--independent of procedure, module, or basic block boundaries--into a contiguous portion of the instruction address space. Such a transformation should benefit any instruction cache organization, since it will move unused or infrequently used code out of line and will also reduce instruction cache conflicts by placing temporally related code segments in adjacent memory locations, hence decreasing the probability of those segments from conflicting in the cache. The fact that these transformations result in fewer taken conditional branches is an incidental benefit for processors that have a taken branch penalty.

Combining Optimizations. Finally, all four optimizations are applied in combination. The four optimizations applied were loop unrolling up to factor of eight, inlining for procedures up to size ten, and both the PDF and FDPR techniques. The results are summarized in Table 1 ; code size increases up to 24% in the worst case, while execution time improves up to 21.5% in the best case.

3. Experimental Framework

Our experimental framework consists of three main phases: optimized compilation with the various options described in the preceding section; trace generation; and microarchitectural timing simulation. All benchmarks are compiled with the IBM xlC compiler for AIX version 4. Traces are collected and generated with the TRIP6000 instruction tracing tool, which is an early version of a software tool developed for the IBM RS/6000 that captures all instruction and address references made by the CPU while

TABLE 2. Benchmark Summary. Taken branches are shown as percent per instruction.

	Baseline		Inline10		Unroll8		PDF		FDPR		Combined	
Benchmark	**Run Length**	**Taken Br**	**Run Length**	**Taken Br**	**Run Length**	**Taken Br**	**Run Length**	**Taken Br**	**Run Length**	**Taken Br**	**Run Length**	**Taken Br**
go	23482056	12.35%	-0.50%	12.35%	-.31%	12.82%	-1.38%	11.46%	0.32%	9.01%	-0.70%	9.06%
m88ksim	88847528	9.06%	-2.77%	8.48%	3.95%	10.76%	-0.30%	8.97%	-0.81%	6.52%	0.56%	8.38%
gcc	22837047	10.61%	-1.06%	10.42%	0.50%	10.76%	-2.41%	9.76%	0.00%	10.25%	-0.74%	8.17%
compress	36416415	11.39%	-12.03%	8.03%	-1.11%	11.38%	-6.43%	10.51%	0.22%	10.25%	-18.36%	7.34%
li	49837753	11.07%	-13.04%	9.21%	-2.91%	11.71%	-0.73%	11.53%	-0.16%	9.39%	-16.84%	8.52%
ijpeg	59822952	12.07%	-1.14%	12.02%	1.87%	13.82%	-0.80%	11.32%	-0.14%	8.14%	-0.27%	10.65%
perl	47802761	9.50%	-2.67%	9.36%	0.10%	9.54%	-0.46%	9.37%	-0.62%	8.50%	-3.62%	8.41%
vortex	58707642	6.52%	-0.11%	6.37%	10.86%	6.12%	-0.27%	6.51%	7.33%	6.39%	7.64%	6.30%
Mean/Total	**390.5M**	**9.96%**	**-4.04%**	**9.22%**	**2.37%**	**10.61%**	**-1.21%**	**9.67%**	**0.84%**	**8.01%**	**-3.18%**	**8.36%**

in user state. Supervisor state references between the initiating system call and the corresponding return to user state are lost; these are generally accepted as being relatively unimportant for the SPECINT95 benchmarks. Finally, the instruction trace is fed to a cycle-accurate microarchitectural simulator that correctly accounts for the behavior of each type of instruction. Our microarchitectural model is implemented using the VMW framework [10].

We simulated two machine model configurations based on the PowerPC 604e microarchitecture, but scaled from a width of four instructions per cycle to 16 instructions per cycle. Furthermore, to focus our experiments on instruction fetch bandwidth and to eliminate secondary performance effects caused by structural dependences and resource conflicts outside the fetch unit, we removed most such structural restrictions. This approach is consistent with that taken by Rotenberg et al. in the original trace cache paper [2]. Specifically, our model implements the following assumptions:

- Branch prediction: 64-way fully-associative branch target address cache (BTAC) and a 512-entry direct-mapped bimodal branch history table (BHT).
- Trace selection: A 64K entry Gag-16 global branch predictor that is capable of making up to three branch predictions in each cycle.
- Trace cache: 512 entries, indexed by fetch address, and 4-way *path associative* via branch bits. Partial matching is not supported; partial squashing is [2].
- Support for a 512 instructions scheduling window.
- Perfect alias detection and store-to-load forwarding
- PowerPC 604e-like primary caches.
- A 3-cycle 256KB 8-way set-associative off-chip cache.
- A 10-cycle perfect level 3 cache

Our trace cache configuration is very similar to the one described by Rotenberg et al. [2]. However, we increased the number entries so the size of the trace cache array matches the instruction cache (32KB of instructions), arguably reaching approximately the same cycle time. Doing so results in a trace cache configuration with 512 entries of up to 16 instructions and 3 branches each, indexed by fetch address and 4-way path-associative based on branch prediction bits. We also use a Gag-16 global multiple branch predictor, identical to the one described in [2], to generate up to three branch prediction bits per cycle. The trace cache fill unit terminates a trace on any of four conditions: the fill buffer contains 16 instructions; the fill buffer contains three branches; the fill buffer contains a computed branch or subroutine return; or the fill buffer contains a system call or trap instruction. The baseline fetch unit terminates fetch on a cache line boundary, predicted taken branch, or a maximum of 16 instructions fetched.

4. Results

We used the SPECINT95 benchmark set as shown in Table 2 , which also summarizes the run length and the percentage of instructions that were taken branches for each of the six compiler configurations. With few exceptions, inlining is the only optimization that significantly reduces pathlength, whereas unrolling is the only one that increases it. The same trend holds for the frequency of taken branches; all optimizations except unrolling slightly reduce the frequency of taken branches.

Effective Fetch Bandwidth. Figure 1 shows the effective fetch bandwidth in terms of the number of useful and

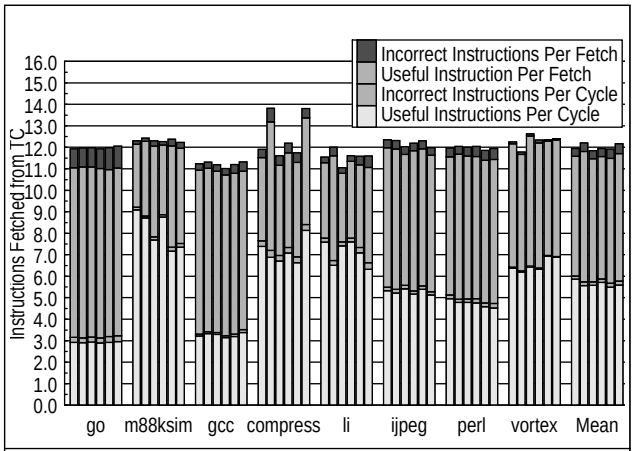

FIGURE 1. Effective Fetch Bandwidth. From left to right, the bars show baseline, inlining, loop unrolling, PDF, FDPR, and combined opt.

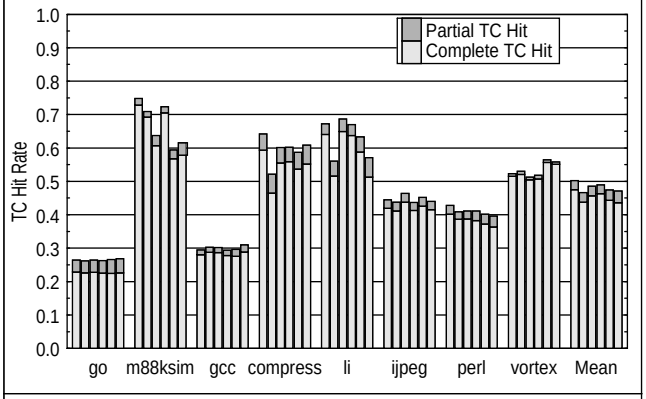

FIGURE 2. Trace Cache Hit Rates. From left to right, the bars show baseline, inlining, unrolling, PDF, FDPR, and combined opt.

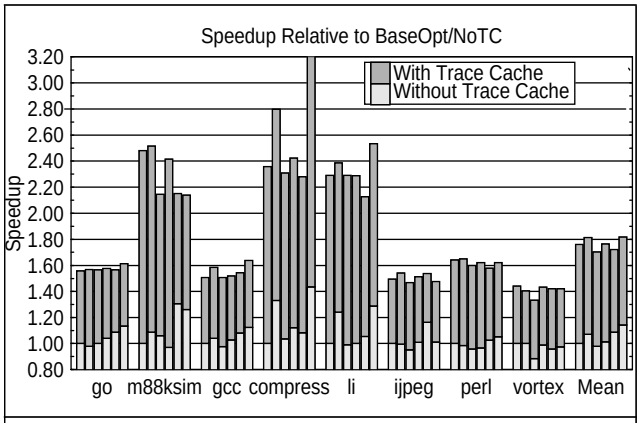

FIGURE 3. Performance Results. From left to right, the bars show baseline, inlining, loop unrolling, PDF, FDPR, and combined opt.

incorrect basic blocks and instructions fetched from the trace cache.The results are shown both per successful trace cache fetch and per execution cycle. On average, the trace cache effective fetch bandwidth does not vary significantly between the various levels of optimization. In general, the number of incorrect instructions fetched increases slightly with more aggressive optimization. Three of the benchmarks (*m88ksim*, *compress*, and *li*) show significant variation in instruction fetch bandwidth per cycle as the optimization level changes; for *m88ksim* and *li* this can be attributed to the corresponding variation in trace cache hit rate. For *compress*, the variation is largely caused by inlining, which eliminates the truncation of trace cache entries caused by subroutine returns.

Trace Cache Hit Rates. Figure 2 shows the both the overall hit rates for the trace cache for each benchmark and configuration, as well as the fraction of each trace cache fetch that was partially wrong (i.e. a partial hit). The latter fraction reflects the inaccuracy of the Gag-16 global branch predictor used to select a trace from the trace cache. Inaccurate predictions lead the fetch mechanism to choose a trace with the wrong branch path out of the trace cache. The low hit rates for *go* and *gcc* can be attributed to their large instruction footprint and unpredictable branch behavior.

Overall, loop unrolling slightly reduces the effective fetch bandwidth and trace cache hit rate. This effect can be attributed to the increased instruction footprint created by the unrolled code (see Table 1). Interestingly enough, the same does not hold for inlining: even though the hit rate is reduced by roughly 5%, the effective fetch bandwidth actually increases. Again, in the case of inlining, the elimination of subroutine return instructions allows better utilization of trace cache entries. This suggests that allowing subroutine returns in the trace cache entries--with associated cost increase in the tag array to hold the return addresses--might be a beneficial hardware optimization. With one exception, the other two optimizations do not significantly affect the trace cache hit rate. In the case of *m88ksim*, FDPR causes a significant drop in the trace cache hit rate. This correlates with a significant drop--roughly 1/3--in the rate of taken branches (see Table 2), which in turn increases conflicts in the Gag-16 branch predictor (see Section 5).

Speedup. Figure 3 summarizes performance results for all of the configurations and benchmarks. First of all, the data clearly show that trace cache provides a significant boost in performance regardless of optimization level. However, the relative benefit of trace cache declines noticeably with increasing optimization level. In some cases (*m88ksim*, *li*) this decline is quite dramatic, and correlates with the decline in trace cache hit rate (see Figure 2), which turn is caused by increased instruction footprint and worse utilization of the branch predictor's pattern history table due to the reduced number of taken branches. In the case of *li*, this decline in trace cache effectiveness is compensated by a significant performance boost from the aggressive optimizations, resulting in overall speedup. However, in the case of *m88ksim*, even though the combined optimization version runs significantly faster without a trace cache, it actually runs slower than the baseline optimized version with the trace cache. We explore this performance anomaly in greater detail in Section 5. The same holds true to a much lesser extent for *ijpeg* and *perl*.

The *compress* benchmark shows the most dramatic variation in speedup. This can be attributed largely to the increased fetch bandwidth (see Figure 1) and reduced pathlength (see Table 2) brought about by inlining. The other optimizations have minor effects when applied individually, but are more than additive when applied in concert.

Sensitivity to Trace Cache Size. Figure 4 summarizes performance sensitivity to trace cache size for the baseline and combined optimization cases. Performance shown is relative to the performance of each benchmark with baseline optimizations applied, running on a processor model without a trace cache. Once again, *m88ksim* demonstrates that aggressive optimizations that increase performance without a trace cache can actually reduce performance with large trace cache. Interestingly enough, performance increases for the small trace caches (32 and 64 entries) but starts to drop

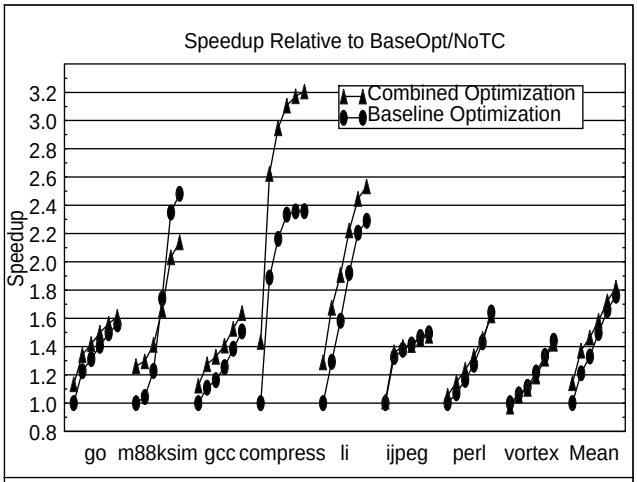

FIGURE 4. Sensitivity to TC Size. Sizes shown are 0, 32, 64,128, 256, and 512 entries.

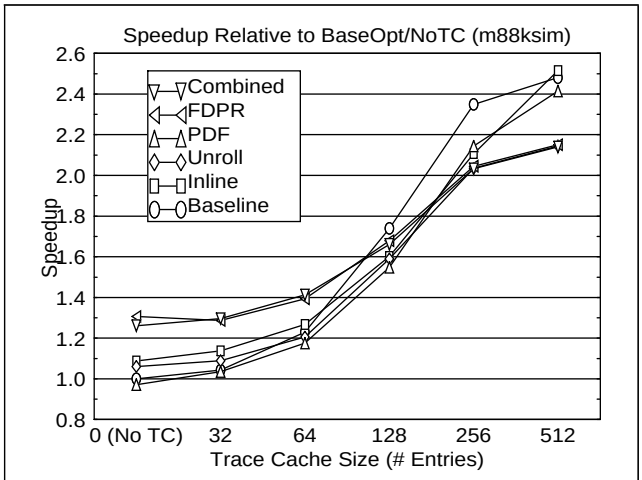

FIGURE 5. Sensitivity of m88ksim to TC Size.

off for larger trace caches. This can be largely attributed to the increase in partial hits shown in Figure 2 (the absolute number of partial hits double between the baseline and combined optimization cases). These in turn are caused by inaccurate predictions from the Gag-16 predictor. The aggressive optimizations, particularly FDPR, which reduces the number of taken branches, are reducing the efficacy of the global predictor by increasing aliasing between entries (since fewer branches are taken, the branch history register used to index into the pattern history table will contain more zeroes and fewer ones, resulting in a worse distribution of branches into the PHT). Both *ijpeg*, *perl*, and *vortex* show the same slight degradation with large trace caches.

On average, however, Figure 4 shows that aggressive optimizations working in concert with trace caches--even small trace caches--can provide significant performance benefits. In fact, over the set of benchmarks shown, the performance of a processor with a given size trace cache running baseline-optimized code is roughly matched by the performance of a processor with half the number of trace cache entries running aggressively-optimized code. Of course, there is a diminishing return with larger and larger trace caches, but the relative benefit of aggressive code optimization is quite significant if area or timing constraints allow only a small trace cache or no trace cache at all.

5. Performance Anomaly

One of the benchmarks in our study (*m88ksim*) demonstrates behavior that is anomalous with respect to the other benchmarks. This is evident from the fact that the baseline and combined optimization plots shown in Figure 4 intersect as the number of trace caches sets increases from 64 to 128. To study this further, we collected detailed information for m88ksim at all optimization levels and trace cache sizes. This data is plotted in Figure 5. The beneficial effect of the FDPR optimization, both on its own and in concert with the others in the combined optimization case, is evident for small trace caches (64 or fewer entries). However, the relative benefit levels off for larger trace caches. The other optimizations are at a clear disadvantage for smaller trace caches, but end up overtaking the FDPR and combined optimization cases at 128 or 256 trace cache entries. The best overall performer ends up being the inline case with a very large trace cache; this follows from the increased instruction working set caused by inlining placing greater demands on the trace cache. However, it is only marginally better than baseline optimization for the 512 entry trace cache, and in fact worse for the 128 and 256 entry trace caches.

The interesting question posed by Figure 5 is the lackluster performance of FDPR optimization with large trace caches. Earlier we postulated that this was caused by increased interference in the pattern history table used to generate branch predictions for trace selection. To verify this, we collected data on PHT interference and plotted it in Figure 6. A PHT update is counted as interference if it meets the following conditions: it must change the state of the counter at the PHT entry, and it must be changed by a branch at a different PC value than the last branch that changed the state of that particular counter. Figure 6 shows the cumulative distribution of interfering PHT updates for PHT entries that are accessed with a fixed number of taken branches in the branch history register (BHR) used to index into the PHT. The smooth curve in the middle shows the expected shape of a uniform distribution over all the PHT entries. The first four optimization cases (baseline, inline, unroll, and PDF) have a roughly uniform distribution of interference. The last two (FDPR and combined) show a significant skewing that results in a 30x increase in interference and three orders of magnitude increase for PHT entries indexed with few (less than 4) taken branches in the BHR. This is a direct consequence of FDPR's aggressive conversion of taken conditional branches to not-taken conditional branches, which results in a paucity of taken branches in the BHR. A summary version of this data is plotted for all the benchmarks in Figure 7; interestingly enough, we see that PHT interference increases significantly (3x on average) for all the benchmarks when FDPR is applied. In most cases, the negative performance effect of this increase in interference

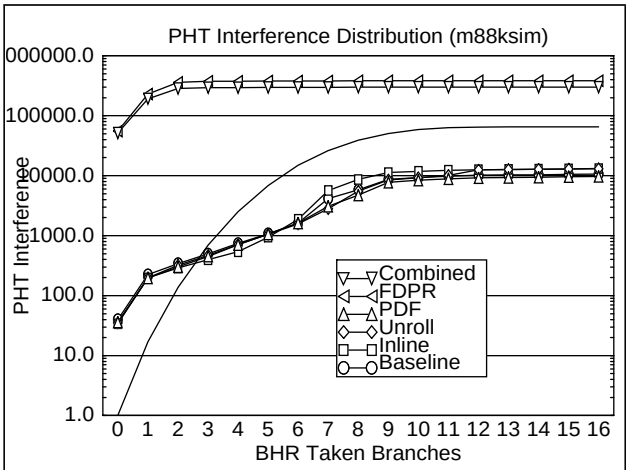

FIGURE 6. PHT Interference Distribution.

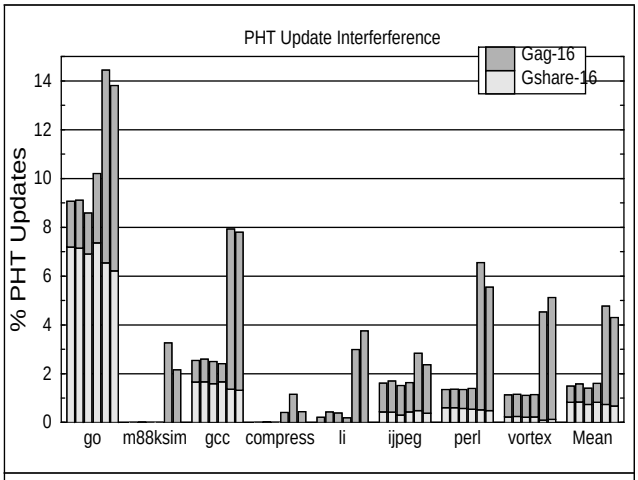

FIGURE 7. Total PHT Interference.

is masked by other factors; it surfaced for us only in the case of m88ksim. One obvious solution to the interference problem is to use an alternative indexing scheme known as gshare [3]; this scheme employs the exclusive-or of the BHR value and the branch PC to index the PHT, and has been shown to improve branch prediction accuracy. Figure 7 also plots the amount of PHT interference experienced with such a scheme. Clearly, the gshare scheme eliminates most of the interference problem and shows hardly any sensitivity to optimization level. However, building gshare indexing--which requires knowledge of the addresses of multiple branch instructions in the trace cache entry before the entry is fetched--into the trace cache fetch path is non-trivial and beyond the scope of this paper. Approximations to gshare indexing have been suggested elsewhere [4], and are probably an appropriate solution to the interference problem that is caused by the proliferation of FDPR-like compiler optimization.

6. Conclusions

In this paper we examine the interaction of aggressive compiler optimization techniques--loop unrolling, automatic procedure inlining, profile-directed feedback, and feedback-directed program restructuring, applied individually and in concert--and the trace cache fetch mechanism. We do so in the context of a high quality, optimizing, production compiler that is in widespread commercial use and a cycle-accurate simulation model based on the PowerPC 604e that implements trace cache. We find that the relative benefit of adding trace cache declines with increasing optimization level. We also find that FDPR optimizations, which removes taken branches based on profile information and improves performance on a processor model without trace cache can actually degrade performance on a processor with trace cache; this can be attributed to increased interference in the pattern history table that is caused by a paucity of taken branches in the branch history register used to index the PHT. A possible solution to the interference problem is to employ an alternative PHT indexing scheme like gshare. Finally, we find that the performance obtained with a trace cache of a given size can be obtained with a trace cache of about half the size by applying aggressive compiler optimization techniques. Our results are in agreement with a contemporaneous study of the same issues using a different architecture and slightly different parameters [11].

References

[1] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of ISCA-22*, pages 333–344, Santa Margherita Ligure, Italy, June 1995.

[2] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of MICRO-29*, December 1996.

[3] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corp, June 1993.

[4] D. Friendly, S. Patel, and Y. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the MICRO-30*, December 1997.

[5] S. Weiss and J. E. Smith. A study of compilation techniques for pipelined supercomputers. In *Proceedings of ASPLOS-II*, pages 105–109, 1987.

[6] S. McFarling. Procedure merging with instruction caches. In *Proceedings of the ACM SIGPLAN PLDI-91*, pages 71–79, Toronto, June 1991.

[7] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 3:168–192, 1971.

[8] W.-M. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of ISCA-16*, pages 242–251, Jerusalem, May–June 1989.

[9] R. R. Heisch. Trace-directed program restructuring for AIX executables. *IBM Journal of Research and Development*, 38(5):595–603, September 1994.

[10] T. A. Diep and J. P. Shen. VMW: A visualization-based microarchitecture workbench. *IEEE Computer*, 28(12):57–64, 1995.

[11] A. Ramirez, J.-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. In *Proceedings of ACM ICS-99*, Rhodes, Greece, June 1999.