On the Effectiveness of Code Reordering Algorithms for theAlpha and IA32 Architectures

TR 97-06-02

Ori Gershony, Jean-Loup Baer and Dennis Lee[*]
Department of Computer Science and Engineering, Box 352350
University of Washington
Seattle, WA 98195

June 30, 1997

Abstract

The impact of instruction cache misses and branch mispredictions on performance is becoming increasingly important for processors that issue multiple instructions per cycle. Mechanisms that address these two sources of performance degradation need to be studied and refined. In this paper, we consider the effects of one such mechanism, namely code reordering algorithms.

We evaluate the performance improvements of three variations of the Pettis and Hansen code reordering algorithm on two instruction set architectures: RISC Alpha and CISC IA32. We show that the algorithms lead to substantial improvements in the fall-through rate of branches which results in decreases in instruction cache miss rates and branch mispredictions. We show that the improvement in fall-through rate and branch prediction is comparable in both ISA's but that the relative improvement in the instruction cache miss rate is higher on the IA32 architecture.

[*]This work was supported in part by NSF Grant CCR-9401689, by Intel Corp. and by a fellowship from National Semiconductor

1 Introduction

Many current and next generation processors use aggressive out-of-order execution techniques to hide the penalty of high latency operations such as data cache misses. Unfortunately, these architectures cannot hide the penalty associated with instruction cache misses and branch mispredictions. Hence mechanisms that address these two sources of performance degradation are becoming increasingly important. In this paper, we consider the effects of one such mechanism, namely code reordering algorithms.

Code reordering algorithms improve performance by taking procedures and basic blocks of a program and reordering them based on profile information to take advantage of specific aspects of the memory hierarchy and the branch control implementation. For modern architectures, this involves mostly increasing the number of fall-through branches. A binary that is reordered this way has several performance advantages over a non-reordered binary. First, since many architectures impose a misfetch penalty even for branches that are correctly predicted but do not appear in the Branch Target Buffer (BTB)[1], code reordering algorithms reduce the misfetch penalty by reducing the number of taken branches. Second, since most modern architectures only store taken branches in the BTB, code reordering algorithms effectively reduces the BTB miss rate because a smaller number of branches need to use the BTB. Third, by placing code that is commonly used together closer in space, code reordering algorithms reduce the number of conflict misses in the instruction cache and increase the utilization of cache lines. Fourth, by increasing the length of uninterrupted code sequences, code reordering algorithms increase the effectiveness of prefetching between the cache and the instruction buffer as well as expand the dynamic instruction window size. Finally, by moving rarely used code to the end of a binary, code reordering algorithms reduce the working set size of a program. This compaction can decrease memory requirements, reduce the Translation Lookaside Buffer (TLB) miss rate, and reduce the traffic on the processor-memory bus.

In this paper, we evaluate the performance improvement of the Pettis and Hansen code reordering algorithm [Pettis & Hansen 90] on two different Instruction Set Architectures (ISA): CISC (Intel IA32) and RISC (DEC Alpha). We show that the algorithm substantially improves the fall-through rate of branches, reduces the instruction cache miss rate, and improves the accuracy of branch prediction. We also show that the improvement in fall-through rate and branch prediction is comparable in both ISA's but that the relative improvement in the instruction cache miss rate is slightly higher on the IA32 architecture.

The rest of this paper is organized as follows. In Section 2 we briefly review related work in code reordering algorithms and describe in detail our implementation of the Pettis and Hansen algorithm [Pettis & Hansen 90]. Section 3 presents our evaluation methodology including our tools, benchmarks and experiments. Section 4 contains the results and analyses of our simulations. We conclude with a summary of our results and suggestions for future work in Section 5.

[1]For example, the MIPS R10000 and the DEC Alpha 21164 impose a once cycle penalty even for correctly predicted branches.

2 The Pettis and Hansen Code Reordering Algorithm

2.1 Related Work

Several code reordering algorithms have been proposed over the past few years. Pettis and Hansen [Pettis & Hansen 90] introduced the algorithm that we implemented for this work. Their algorithm yields an impressive reduction in the instruction cache miss rate and does not use inlining which can increase the size of the binary. Hwu and Chang [Hwu & Chang 89] proposed a similar algorithm, with inlining, that performs layout of traces (basic blocks which tend to execute in sequence) instead of basic blocks. McFarling [McFarling 89] takes a slightly different approach: instead of trying to position related basic blocks close to each other, he tries to position them in non-conflicting cache locations. His algorithm is sensitive to the size of the instruction cache and does not give a comparable increase in the branch fall-through rate, but can yield the lowest instruction cache miss rate. In a similar vein, Hashemi et al. [Hashemi, Kaeli, & Calder 97] use Pettis and Hansen as a base and extend it through cache line coloring to avoid conflicts between concurrently executing procedures. Their algorithm is cache size and line size specific. They report a reduction of 17% over Pettis and Hansen in instruction cache miss rate. Calder and Grunwald [Calder & Grunwald 94] address specifically the branch alignment problem and give two algorithms that improve on Pettis and Hansen's for the reduction of penalties due to branch mispredictions.

Spurred by the advances in hardware based two-level branch predictors [Yeh & Patt 91], Krall [Krall 94] and Young and Smith [Young & Smith 94] have proposed schemes which, by profiling patterns, can recognize correlated branches. Prediction performance can be improved at the expense of increased code size through duplication of some basic blocks. We are not aware of any study that measures the impact of such schemes on the instruction cache miss rate.

Finally, we should note that control flow predictions based on profiling runs are generally good [Fisher & Freudenberger 92]. Our experiments confirm this observation. The results that we will report in Section 4 are based on input testing sets different from the learning input sets.

2.2 Pettis and Hansen algorithm

The Pettis and Hansen code reordering algorithm consists of two main parts: procedure positioning and basic block positioning. Basic block positioning is done separately for each procedure so code from different procedures is not mixed. We describe this as the *local* approach in Section 2.3. Depending on whether we keep the order of procedures unchanged or we apply procedure positioning, there are two possible local reorderings: 1) basic block only (*bb only*) where the original order of procedures is left unchanged, and 2) procedure positioning in addition to local basic block positioning (*proc-block*). It is also possible to apply Pettis and Hansen's basic block positioning algorithm to the whole binary, hence mixing code from different procedures. We call this the *global* approach and describe it in

Section 2.4.

2.3 A Local Algorithm

2.3.1 Procedure Positioning

The procedure positioning algorithm takes an undirected call graph as input, and produces an ordered list of procedures as output. In the call graph, vertices correspond to procedures, and edge weights designate the dynamic call frequency obtained by the profiler. The heuristic used by the algorithm is "closest is best", meaning that if one procedure calls another frequently, we want them to end up close to each other in the final code. This strategy helps improve performance in several ways. First, it helps avoid some instruction cache conflicts between procedures that should often be in the cache at the same time. Second, it reduces the distance of branches, which can have an impact on the length of the branch instructions (such as with the Intel IA32 architecture), or on the type of instructions used for the branches (such as with the DEC Alpha architecture). Finally, this heuristic can help reduce the number of pages in the working set of an application, hence decreasing memory requirements.

The algorithm takes a greedy approach by examining all the edges in order of decreasing weight. At each stage, the two nodes connected by the edge of highest weight are merged. A merged node contains an ordered list of all the procedures that compose it. Hence every node in the graph is either a primitive node which corresponds to exactly one procedure, or is a merged node which contains an ordered list of procedures.

The primary observation made by Pettis and Hansen is that when merging two nodes, at most four different configurations need to be considered, and that these configurations are obtained by the binary choices of reversing the first node and reversing the second node. For example, when merging A and B, we only need to consider the following four configurations: AB, $A_{reverse}B$, $AB_{reverse}$, and $A_{reverse}B_{reverse}$. Note that each of these configurations has a dual which corresponds to a node with the reverse configuration, that is, a node whose procedure list is reversed. For example, the dual of AB is $B_{reverse}A_{reverse}$. However, the duals need not be considered at this stage because the procedure ordering in the merged node C can be reversed when C is merged with another node if necessary. The best configuration is chosen via a rank function that takes the "closest is best" strategy into account. In our implementation, we followed the original paper and used the sum of the edge weights for all adjacent procedures as the ranking function.

2.3.2 Basic Block Positioning

Once the procedures have been positioned, either by keeping the original ordering or by applying the above algorithm, the basic block positioning algorithm tries to improve performance by making as many branches fall through as possible. We expect that block reordering will increase the average number of instructions executed per cache line, and reduce the in-

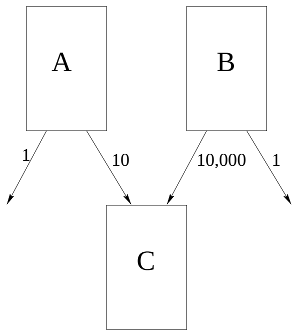


Figure 1: *An example where depth-first traversal could produce a bad layout if C rather than B is selected to be close to A.*

struction cache miss rate. We also expect that it will reduce the number of entries in the BTB (because the BTB does not allocate entries for fall-through branches), which in turn reduces the BTB miss rate. Moreover, replacing correctly predicted taken branches by correctly predicted non-taken branches is also a benefit for some microarchitectures. Finally, block reordering should result in longer uninterrupted, control-flow wise, code sequences and in effectively expanding the dynamic instruction window size and in providing more opportunities for code prefetching.

One major difference between the basic block positioning algorithms and the procedure positioning algorithms is that basic block positioning algorithms deal with directed graphs. In other words, since it is important for most branches to fall through, the final ordering of basic blocks must depend on the order of the nodes in the graph, and not just their distance from each other.

Pettis and Hansen proposed two basic block positioning algorithms in their original paper. The first algorithm performs a simple depth-first traversal of the CFG (Control Flow Graph), giving preference to edges with higher weights. This algorithm, however, doesn't always make great choices. For example, suppose that both basic blocks A and B have control transitions into C, with frequencies of 10 and 10,000 respectively (see Figure 1). If A is encountered first in the depth-first traversal, then C may be placed immediately following A, even though a much better layout would place C immediately following B. The second algorithm overcomes this problem by considering all the edges in order of decreasing weight. This algorithm creates ordered lists of basic blocks using a bottom-up traversal, keeping the direction of edges in the original CFG. For this paper, we implemented the second algorithm.

2.4 A Global Algorithm

The approach described in Section 2.3 has very clean semantics, and is especially appealing from an implementation perspective, because code from different procedures is not mixed. Such a local approach, however, does not take advantage of the full potential of code reordering algorithms. Mixing code across procedures can significantly reduce the working set of an application. We thus consider a global algorithm where the bottom-up basic block positioning algorithm is used across procedures. In other words, the input basic block CFG is that of the whole program and not just a single procedure. In some sense we effectively inline part of a procedure where it is used most often. However, contrary to conventional inlining, we do not replicate code and the "inlining" occurs only once per basic block.

3 Methodology

3.1 Tools and the Measurement Process

In practice, there are two main approaches for applying code reordering algorithms to program binaries: as an optimization phase of the compiler, or as a post-compile optimization phase using a binary rewriting engine. The first method requires the source code for the compiler and may require that the implementation be compiler specific. However, other phases of the compiler would have the opportunity to take advantage of the profile information to generate better code. We chose to use the second method, binary rewriting, to share the same reordering code across two very different platforms running two different compilers (i.e., cc on the DEC Alpha platform and MSVC on the IA32 platform). The binary instrumentation tools that we use are ATOM [Digital Equipment Corporation 94] running on the DEC Alpha and Etch [Romer, et. al. 97] running on Pentium and Pentium Pro.

The measurement process consists of three phases:

1. We instrument the applications to record profiling information. We then run the resulting binaries to produce a CFG where edge weights correspond to frequencies of control transfer. The CFG and associated weights are gathered from a *learning* input set for each application.

2. We use the profiling information to generate the optimized layout according to one of the versions of the Pettis and Hansen algorithm described in the previous section.

3. We measure the effectiveness of the optimized layout by instrumenting the binary and simulating a machine running the optimized binary. These runs are performed on a *testing* input set different from the learning input set.

Note that we never produce a binary with the new layout, because we don't have a tool that modifies the layout of a binary on the Alpha[2]. Instead, the simulator first translates the original addresses to new addresses as if the binary were reordered. This means that our measurements are not perfectly accurate because we don't modify the control transfer instructions to reflect the new layout but we believe that the error is a small, second order effect.

[2]Etch can produce modified binaries on the IA32, but we chose not to use these so that we would use the same methodology on both platforms.

3.2 Benchmarks

Table 1 shows some statistics for the six SPECInt95 benchmarks used for this study. The same source files, learning and testing input sets, were used on both the IA32 and the Alpha, and they were compiled by MSVC 4.2 and Digital Unix cc 3.11 respectively. Measurements were taken only for user activity.

	Executable Size in KBytes		Instr. Executed learning/testing in Millions		Control Transfers learning/testing in Millions		Icache Miss Rate learning/testing	
Applic.	IA32	Alpha	IA32	Alpha	IA32	Alpha	IA32	Alpha
gcc	1478	2195	247/308	349/432	62/77	60/74	3.96%/3.90%	4.99%/4.97%
li	149	172	512/559	815/895	143/156	158/172	1.45%/1.44%	2.49%/2.63%
ijpeg	257	328	649/580	1018/910	119/105	117/104	0.00%/0.00%	0.03%/0.03%
compress	163	106	210/414	399/784	48/97	48/98	0.13%/0.00%	0.00%/0.00%
perl	414	500	12/1057	13/1025	2.7/242	2.3/192	3.91%/5.43%	4.69%/6.28%
m88ksim	223	300	82/418	76/391	14/69	14/70	0.01%/0.00%	4.11%/4.14%

Table 1: *Benchmarks used for this study. The Icache miss rates are for a direct mapped 8K cache with 32 bytes line size. Miss rates of less than 0.01% are shown as 0.00%. Note that the testing set in perl executed 100 times more instructions than the learning set, compress testing set executed twice as many instructions, and the testing set in m88ksim five times as many.*

Table 1 shows that the Alpha RISC architecture generally requires more instructions to accomplish a given task than the IA32 CISC architecture (*gcc, li, ijpeg,* and *compress*). In all cases but one (learning set of *compress*), the Alpha instruction cache miss rates are higher than the IA32's for the 8 KB 32 byte line instruction cache size. Except for *li*, this trend will be consistent even when we vary cache sizes, line sizes, and reordering algorithms. We expect that the IA32 would have the better miss rate since its average instruction size is smaller than the Alpha fixed 4-byte instruction size: more instructions can fit in a cache line and a larger text in the cache itself. An extreme example is *m88ksim* where the miss rate is quite high for the Alpha (over 4%) but negligible for the IA32 even though the IA32 executes more instructions.

While instruction cache miss rates depend to some extent on the ISA, the number of branches depend on the program. We expect that the absolute number of branches be very similar for each application on both architectures. This is indeed the case, with differences of less than 3% on four benchmarks out of six (*gcc, ijpeg, compress,* and *m88ksim*). The outliers are *li* where the Alpha has 9% more branches and *perl* where the IA32 executes 14% more branches. We will discuss these statistics further in Section 4.

3.3 Measurements

Recall that one of the goals of the code reordering algorithms is to increase the number of instructions that fall through hence improving the instruction cache hit rate and branch prediction. We will examine how well this goal is realized by measuring the fall-through rate of control transfer instructions.

To compare the effectiveness of the reordering algorithms on two different ISA's, we assume that the instruction caches and the branch control mechanisms are the same for both architectures. Unless otherwise specified, the instruction cache will be an 8 KB direct-mapped cache with 32-byte line size. The branch architecture will use a 64 entry 4-way set-associative BTB and a 512 entry gshare Pattern History Table (PHT) [McFarling 92]. To estimate the improvement in performance, we measure:

- The instruction cache miss rate.
- The reduction in branch misfetches: A branch misfetch occurs when there is a delay in identifying an instruction as a branch, or when a correctly predicted branch has to wait for its target address to be calculated.
- The reduction in branch mispredicts: A branch mispredict occurs when the behavior of a conditional branch is predicted incorrectly by the PHT. or when an indirect jump has the wrong address in the BTB.

Our expectations are that the reordering algorithms will reduce the cache miss rate by better utilizing the cache lines and avoiding some conflict misses, reduce the BTB miss rate by decreasing the number of branches that require BTB entries, and reduce the number of branch mispredicts by decreasing the harmful effects of aliasing in the PHT.

We combine the results of our measurements by assigning penalty cycles to instruction cache misses, BTB misfetches, and branch mispredicts, and computing the number of cycles saved in the optimized layouts. We recognize that this is only an approximation of the performance improvement because other factors, such as the lengthening of uninterrupted code sequences and using fewer entries in the TLB could add to the improvements. Moreover, a much more detailed simulation would be needed to obtain the exact CPI and execution time of the benchmarks. Nonetheless, by estimating a CPI as reported by other sources and by knowing the number of instructions executed, we can get a good approximation of the potential benefits due to code reordering.

4 Results

We measured the performance of the three variations of the Pettis and Hansen algorithm using the metrics proposed in Section 3. Section 4.1 discusses the control transfer characteristics of the different layouts. These characteristics point towards performance improvements

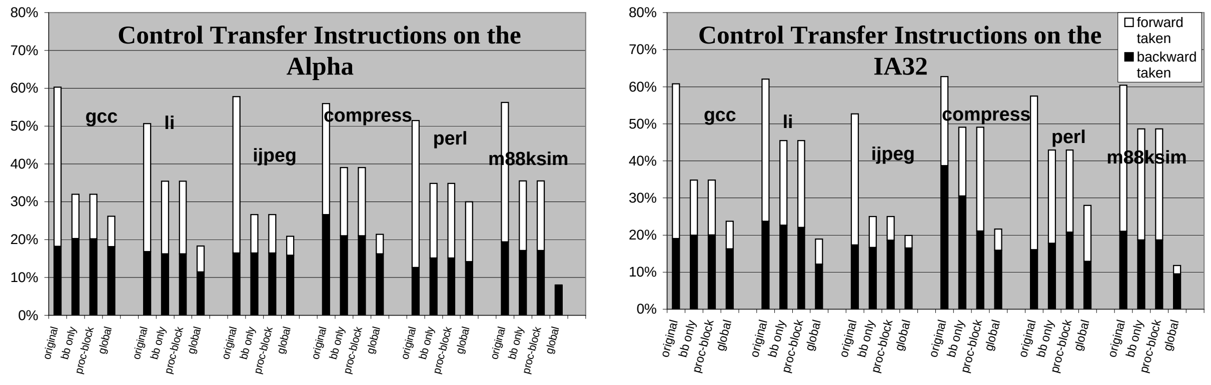


Figure 2: *Percentage of control transfer (forward and backward) instructions that are taken on the original and the 3 variations of the reordering algorithm. Note the large decrease in forward transfers (in white) for the reordered layouts.*

in terms of instruction cache hit rates (Section 4.2), better branch prediction (Section 4.3), and better utilization of the interface between the instruction cache and the processor fetch and decode unit. We combine the first two metrics in Section 4.4 where we present the reduction in the cycles wasted due to instruction cache misses, BTB misfetches, and branch mispredicts.

4.1 Control Transfer Characteristics

The Pettis and Hansen basic block positioning algorithm improves performance by maximizing the number of control transfer instructions that fall through (Section 2.3.2). Control transfer instructions include branches and jumps which transfer control within procedures, and calls and returns which transfer control between procedures.

Figure 2 shows the percentage of control transfer instructions that are taken with the different code layouts. In the figure, backward-taken control transfers are in black, and forward-taken control transfers are in white. As expected, the percentage of taken control transfers is highest with the *original* layout, and lowest with the *global* layout. The two local layouts behave in the same manner because the procedure reordering algorithm does not take the direction of calls and returns into account.

Note that almost the entire reduction in taken control transfers is due to a decrease in the percentage of forward taken control transfers. The number of backward taken control transfers decreases by very little if at all. Backward taken control transfers often define the boundary of loops and are thus hard to eliminate without loop unrolling.

Figure 2 also confirms our expectation that differences in ISA should not impact the occurrences and directions of control transfer instructions: the graphs for the Alpha and the IA32 are very similar.

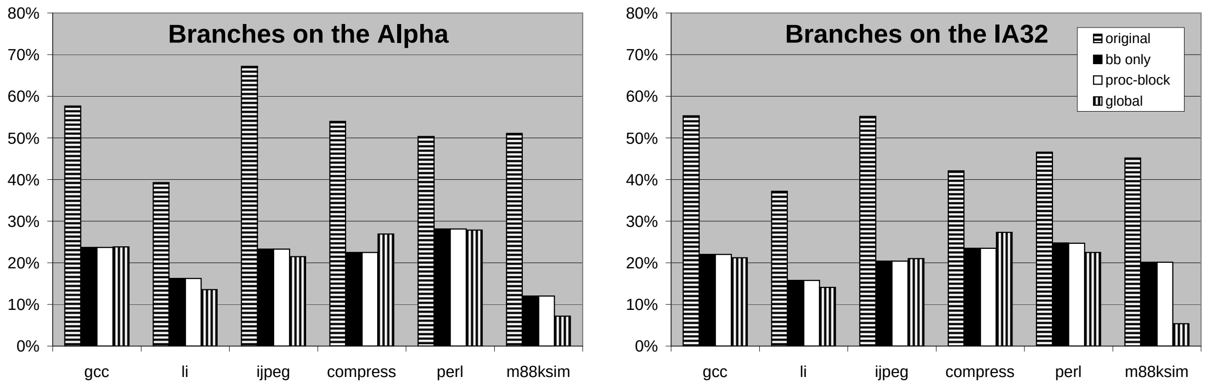


Figure 3: *Percentage of branches that are taken. Notice the large reduction, a factor from 2 to 6 depending on the benchmarks. As expected, this reduction is independent of the ISA.*

Figure 3 shows the same type of information as Figure 2 but consider only branches. It indicates that code reorganization achieves a very important reduction in the proportion of branches that are taken: from a factor of almost 2 in *perl* to a factor of above 6 in *m88ksim*. We will elaborate more on the effect of code reorganization for branch prediction in Section 4.3.

Returning to Figure 2, we see that the number of control transfer instructions is smallest in the *global* layout. This property of the *global* layout is a direct consequence of the one-time "inlining" of some procedures. Comparing Figure 2 and Figure 3, we see that the reduction in control transfer instructions between the *global* layout and the *local* layouts is due to a reduction in control transfers between procedures since the reduction in branches is similar for the three reordered layouts.

Figure 3 puts to rest a concern with the *global* layout, i.e., that it may scatter the basic blocks belonging to a single procedure into discontinuous regions in the address space. This scattering would reduce the number of fall-through branches and hurt branch prediction. The figure shows that mixing code from different procedures in the *global* layout does not obstruct the control flow inside individual procedures.

Another measure of the success of the code reordering algorithms is the increase in the length of code sequences uninterrupted by a transfer of control. Longer uninterrupted code sequences are especially useful for wide superscalar architectures since they result in better utilization of the interface between the instruction cache and the fetch and issue mechanisms[3]. Figure 4 shows the cumulative distribution of the lengths of uninterrupted code sequences for the three code layouts (*bb only* and *proc-block* yield the same results and are called *local* in the figure) on all benchmarks. The horizontal axis designates the number of instructions in an uninterrupted sequence, and the vertical axis designates the percentage of instructions that belong to a code sequence of length equal to or less than that indicated

[3]We will not be able to quantify this increase in terms of saved cycles because it would require simulators for both architectures which are much more detailed than those we have been using.

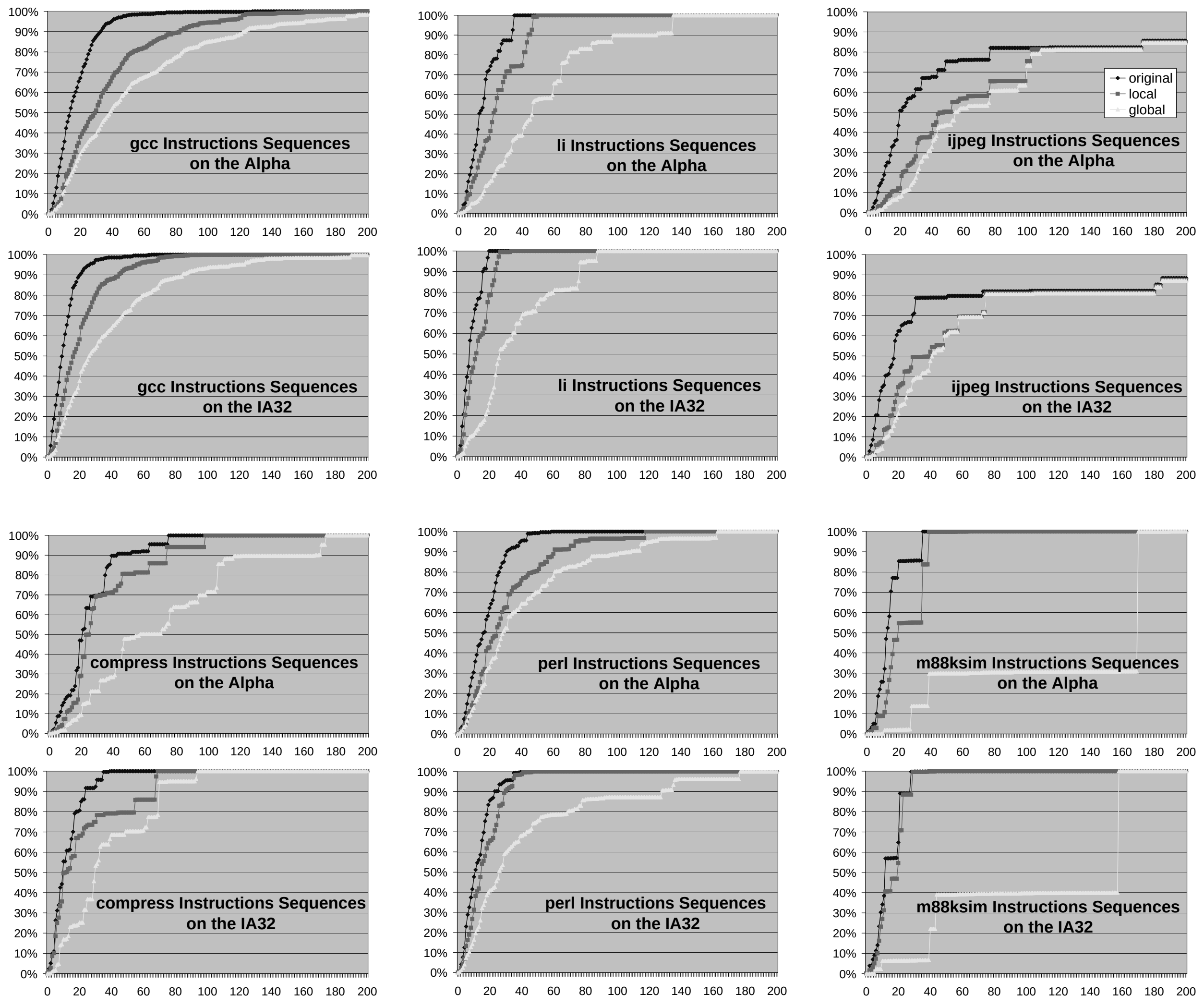


Figure 4: *Cumulative distribution of the lengths of uninterrupted code sequences for the three code layouts (bb only and proc-block yield the same sequences and are plotted as local). For example 80% of the code sequences in the original gcc on the Alpha are of length 20 or less while in the global layout they are of length 90 or less.*

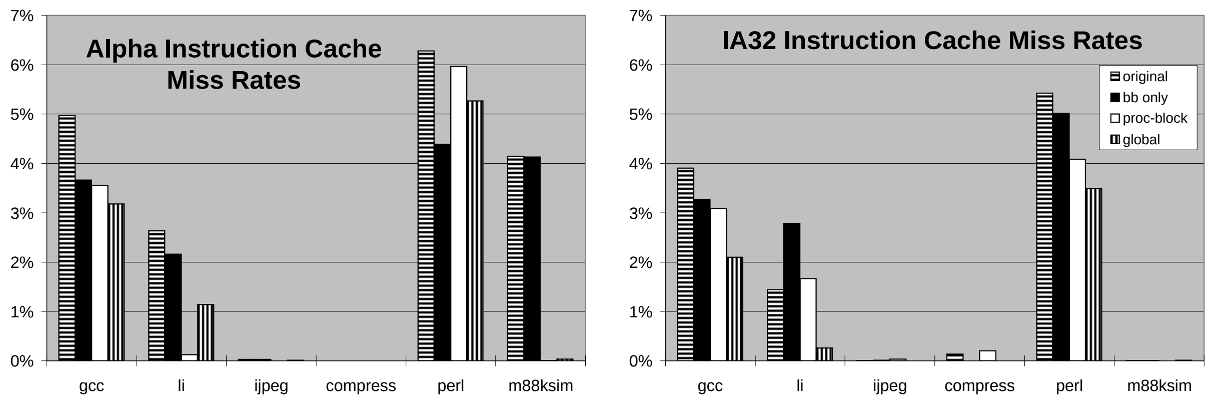


Figure 5: *Instruction cache miss rates for a direct-mapped 8K cache with 32 byte lines for the four layouts.*

by the horizontal axis. Code sequences from the *global* layout are always significantly longer than those from the *original* code. The local layouts have significantly longer sequences only on some of the benchmarks such as on *ijpeg* and *gcc*, but not on *compress*, *li* and *m88ksim*. These applications have frequent control transfers between procedures so procedure local basic block code positioning does not increase the length of uninterrupted code sequences.

Figure 4 also shows that in general code sequences are longer on the Alpha ISA. This is not surprising in light of the fact that it takes more instructions to perform the same task (recall Table 1). Note however that *perl* and *m88ksim* which execute roughly the same number of instructions on each architecture have comparable lengths of uninterrupted code sequences.

4.2 Instruction Cache Miss Rates

Reordering code results in improved instruction cache hit rates for two reasons. First, the program uses more of each cache line because of the longer uninterrupted code sequences. Second, the program experiences less conflict misses, because segments of code that are frequently used together are likely to be placed close to one another.

Figure 5 presents the instruction cache miss rates for a direct-mapped 8 KB instruction cache with 32-byte lines with the different code layouts. Two of the benchmarks, *ijpeg* and *compress*, have insignificant miss rates and won't be discussed any further. One benchmark, *m88ksim*, shows a large miss rate for the Alpha ISA and an insignificant one for the IA32. When the *proc-block* and *global* reorderings are applied to the Alpha version of *m88ksim*, the miss rates become close to zero. We believe that the working set of *m88ksim* just fits in the cache in the IA32 case independently of the layout but that it is slightly too large to fit without modifications in the Alpha case. For the Alpha, reordering procedures or blocks belonging to different procedures removes conflict misses and allows the working set to be resident in the cache.

The three other benchmarks (*gcc, li, perl*) have significant miss rates on both architectures. We expected that the three reordering algorithms would all reduce the miss rates and that the largest reduction would be for *global* followed by *proc-block* and *bb only* in that order. For *gcc* on both ISA's and *perl* on IA32, this is indeed the case. However, code reodering is only a heuristic and does not work perfectly in all instances. While *global* is always better than *original*, it is not as good as *proc-block* on *li* and *bb only* on *perl* for the Alpha. Moreover, the *proc-block* and *bb only* code reorderings of *li* on the IA32 lead to worse miss rates. The strange behavior on *li* is also visible when we apply the reorderings to the learning set so we cannot blame the failure of the algorithms to the differences between the learning and testing data sets. A possible explanation for *li*'s high miss rate on the IA32 for the two local layouts is a conflict between commonly used instructions in different procedures that are called for all input data sets, a conflict that did not exist in the original layout[4].

Overall, though, if we look at the totality of the results, the improvement in the instruction cache miss rate with the *global* layout is very significant; for example improvements of over a factor of two are achieved on both architectures for *li* and of almost a factor of two for *gcc*. Improvements for *perl* are not quite as large but are still important. *proc-block* and *bb only* also result in marked improvements. Generally, the improvements on the IA32 are relatively larger than those on the Alpha.

Independently of the layouts, increasing the cache capacity will always reduce the number of misses and increasing the cache line size, up to a reasonable size, is also known to be beneficial. Reordering algorithms accelerate the rate of improvement since the working set of the application can fit in a smaller cache and the lengthening of uninterrupted code sequences results in a better utilization of cache lines. Thus, it is not surprising to see cache miss rates be reduced with increasing cache capacities (Figure 6) and for a given cache size, with increasing line sizes (Figure 7), for all layouts.

As noted in Table 1, it takes less IA32 instructions than Alpha instructions to execute the same task. The average size of an IA32 instruction is also smaller than the average size of an Alpha instruction, and hence more instructions can fit in each cache line and in the whole cache. The combination of these two factors should result in a better utilization of the cache in the IA32 when we increase cache capacity and when we increase the line size.

Let us define the relative benefit of a layout L as the miss rate for the original layout divided by the miss rate for L. We have computed the relative benefits of the three layouts over the range of cache sizes shown in Figure 6 for *gcc li* and *perl*. Taking the *global* layout as the more consistent example, the relative benefits peak for both ISA's around 16 or 32K caches with a better benefit for the IA32. Similarly, larger cache lines (128 or 256 bytes) yield better relative benefits (computed from the data used for Figure 7) with the larger "effective size" of the cache lines on the IA32 contributing to its slightly higher relative benefit rate.

[4]It is also possible that like all SPEC95 benchmarks, *li*'s original layout is optimized for instruction cache behavior. This could explain also the relatively poor performance of *proc-block* on Perl for the Alpha.

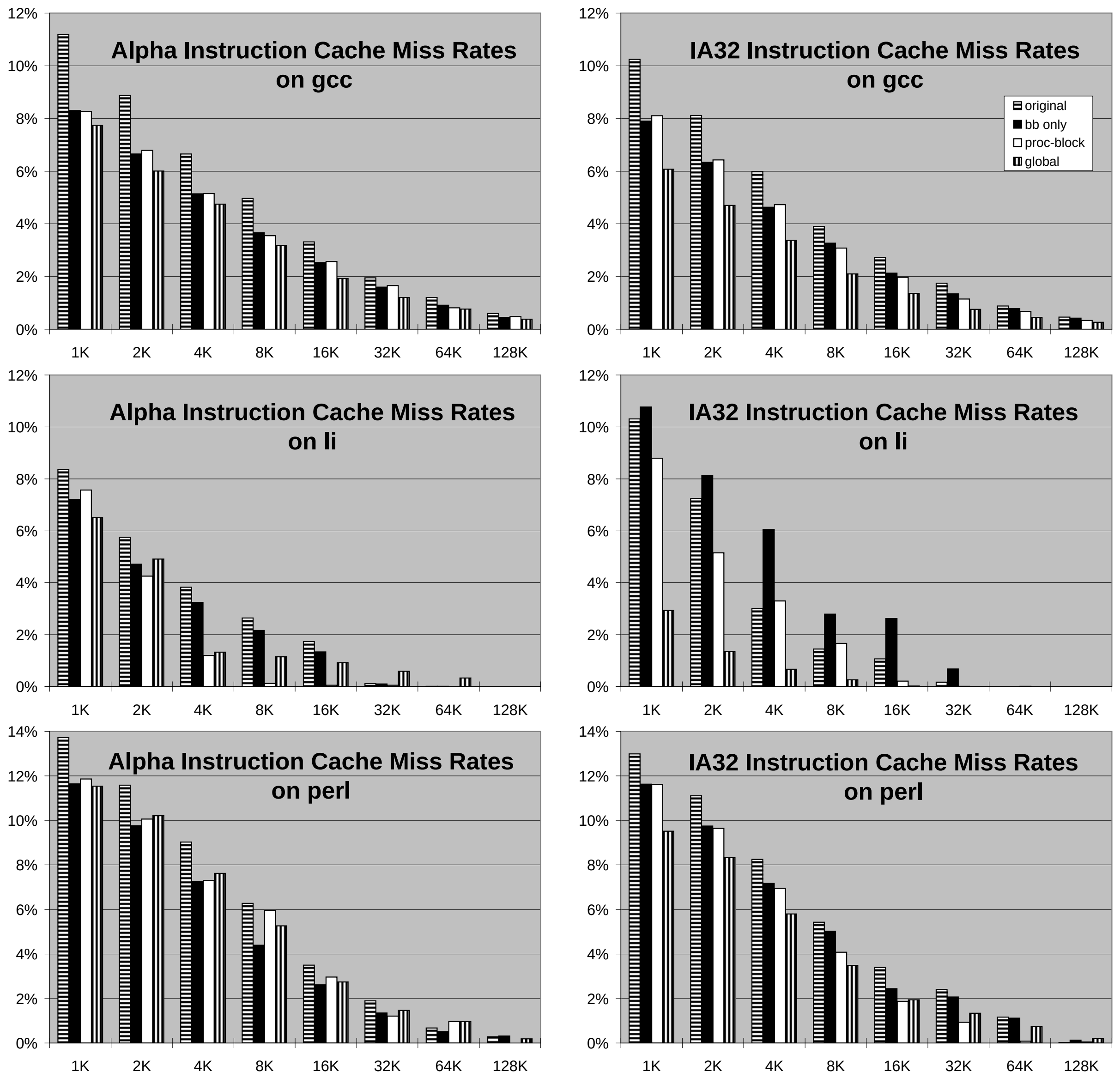


Figure 6: *Instruction cache miss rates for gcc, li, and perl for various cache sizes with 32 byte cache lines.*

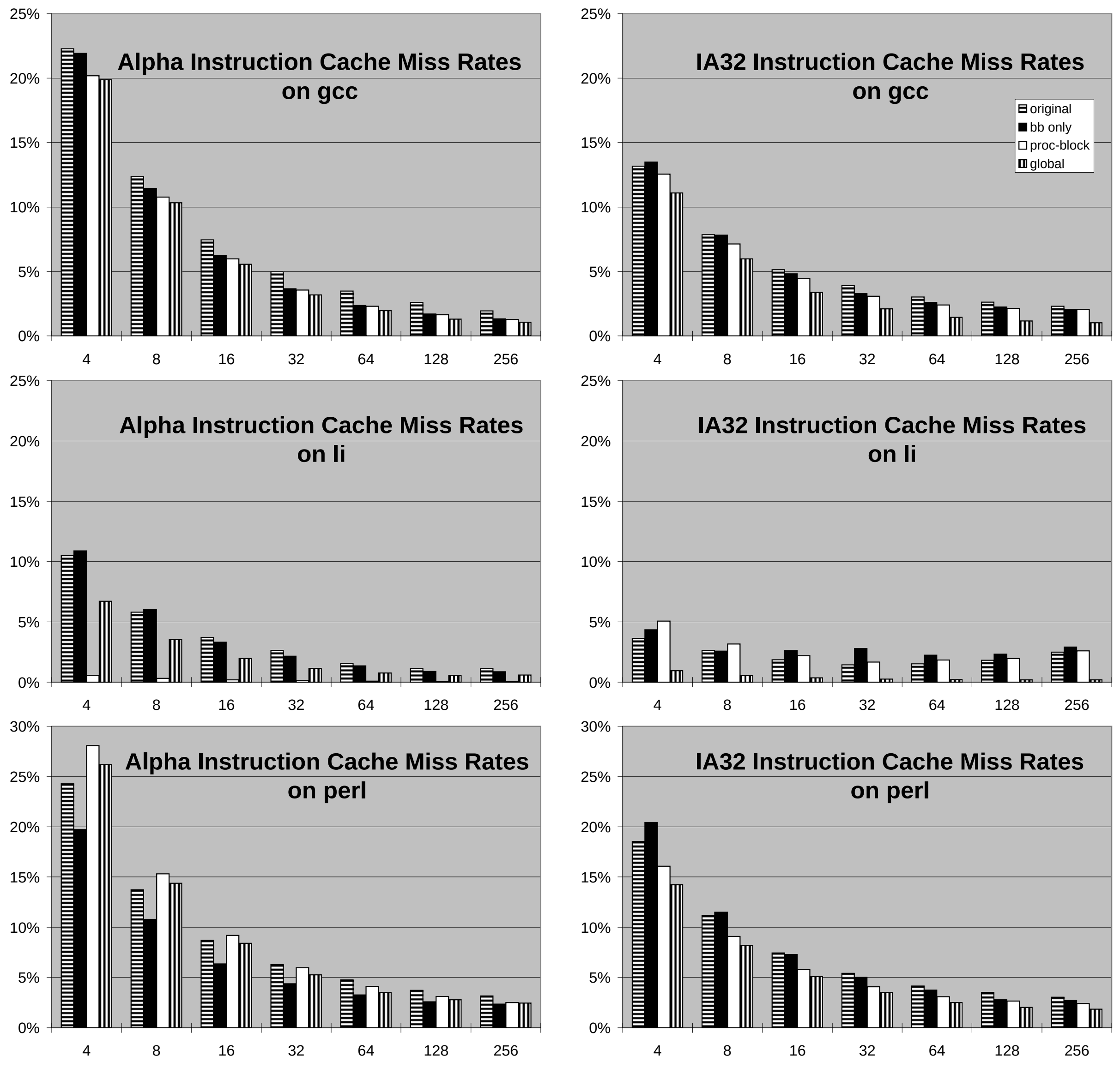


Figure 7: *Instruction cache miss rates for 8K caches with various cache line sizes.*

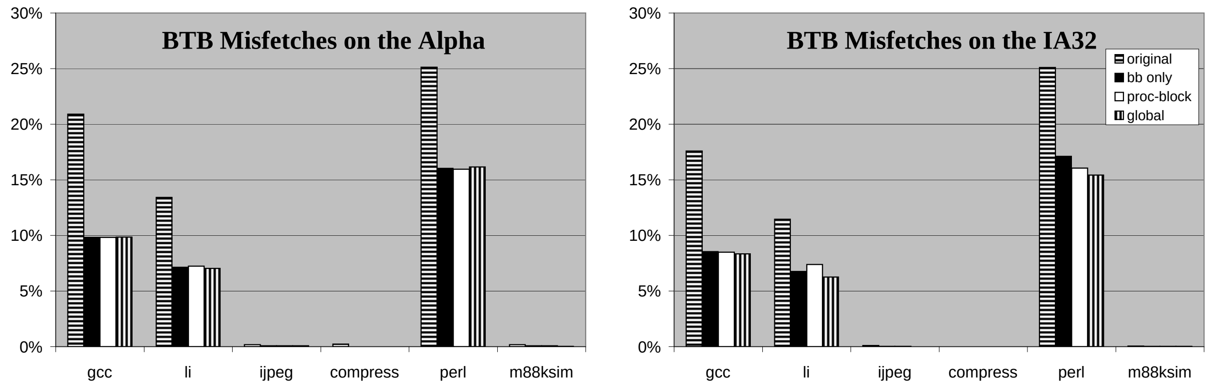


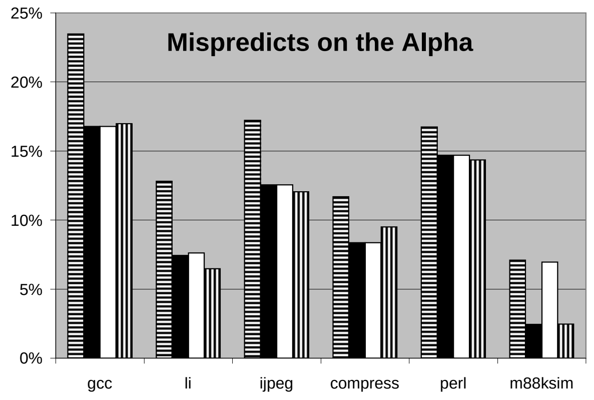
Figure 8: *Misfetch rates for a 64 entry 4-way set associative BTB. The three layouts reduce misfetches by almost a factor of 2 for those benchmarks that have large misfetch rates in the original code.*

4.3 Branch Prediction

The complexity and implementation sophistication of branch prediction mechanisms are far apart in the recent Alpha and IA32 implementations [Bhandarkar 97]. We chose to simulate mechanisms midway between those implementations to evaluate the effects of the code reordering algorithms. The gshare configuration used in our simulations is more complex that the single PHT, associated with cache lines, of the Alpha 21164 and less complex than the scheme present in the Pentium Pro [Microprocessor Report 95, Yeh & Patt 91].

Code reordering algorithms should improve on the use of the BTB because the BTB only stores target addresses for branches that are taken. Increasing the number of non-taken branches should leave more room in the BTB for those that are taken. The number of branch misfetches, i.e., those branches for which there is a delay in identifying an instruction as a branch, or when a correctly predicted branch has to wait for its target address to be calculated, should therefore diminish. Figure 8 which displays the BTB misfetch rates confirms that fact. The misfetch rates of the benchmarks that have large misfetch rates in their *original* layout are substantially reduced –approximately a factor of two on both architectures. Note that the misfetch rates for the three reorderings are very similar for all three layouts and are also similar across architectures.

While the reduction in the number of BTB misfetches is important, a decrease in the number of branch mispredicts can improve performance even more, because the number of issue slots lost for a branch mispredict is much higher than the cost of a BTB misfetch. Recall that a branch mispredict occurs when the behavior of a conditional branch is predicted incorrectly by the PHT or when an indirect jump has the wrong address in the BTB. Figure 9 shows the branch mispredict rates for our simulated mechanism. The mispredict rates are highest on the original layout, generally the same for the three reorderings, and quite consistent across the two ISA's. Differences in some cases occur because of aliasing in the PHT. Overall though, improvements range from a few percent as in *perl* to about a 50%



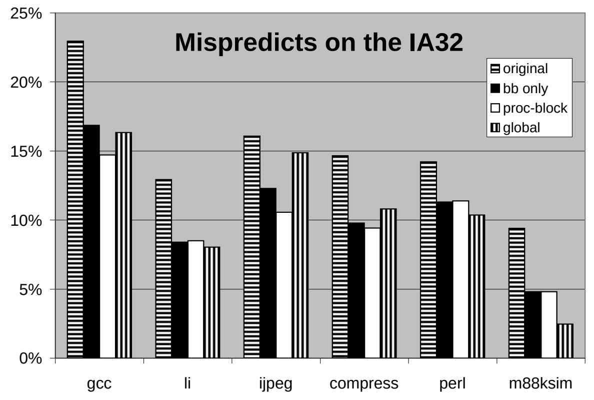


Figure 9: *Branch mispredict rates for a 64 entry BTB and a 512 Entry PHT.*

decrease as in *li*.

This experiment confirms the observations of Section 4.1 that ISA's should have similar branch behavior and that the *global* algorithm does not have an adversary impact on transfers of control.

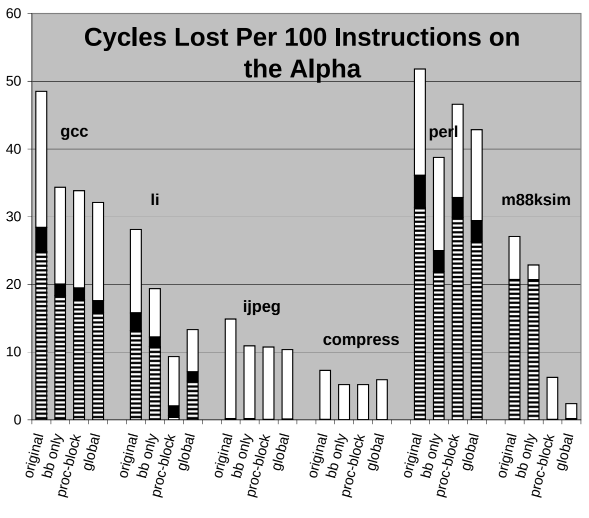
4.4 Estimating the savings in cycles

We have shown that control transfer instructions in the *local* and *global* Pettis and Hansen layouts have higher fall-through rates, which reduce the instruction cache miss rates, BTB misfetch rates, and branch mispredict rates. In this section we combine these reductions into a single metric which estimates the number of cycles saved per 100 instructions.

We use a simple machine model where the cost of a miss in the instruction cache is 5 cycles, the cost of a BTB misfetch is 1 cycle, and the cost of a branch mispredict is 5 cycles. While this model does not accurately simulate modern architectures with complex out-of-order execution designs, it is sufficient to give us a rough idea of the improvement in performance one can expect to see. Note that these penalties are not the same as the ones in the Alpha 21164, or the Pentium Pro. In particular, the penalty for a mispredicted branch on the Pentium Pro is 10-15 cycles – quite a bit more than our choice of 5 cycles[5], but the branch mispredict rate on the Pentium Pro is also lower than the one we gathered in our simulation since we simulated a simpler branch prediction mechanism.

Figure 10 shows the number of cycles lost per 100 instructions due to instruction cache misses, branch mispredicts, and BTB misfetches when we apply the costs defined above. In all cases, reordering algorithms yield benefits. The benefits are larger when both the instruction miss rates and the branch mispredicts are reduced, as for example in *gcc*. Our simulations are not detailed enough to give precise performance improvements. However, we can estimate them with extreme caution by dividing the number of cycles lost reported

[5] 5 cycles corresponds to the branch mispredict penalty on the Alpha 21164



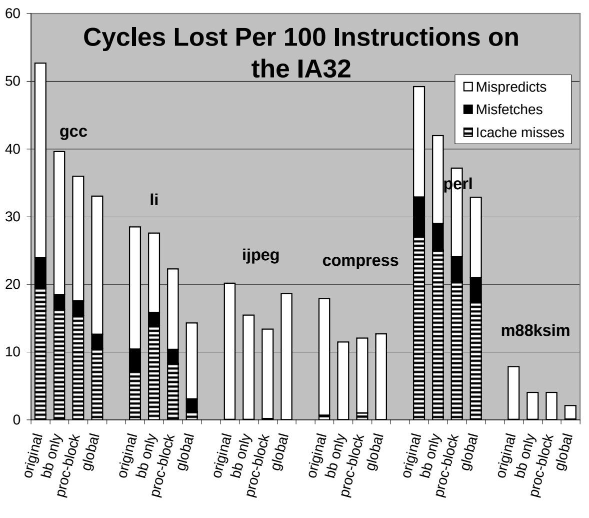


Figure 10: *Cycles lost per 100 instructions due to branch mispredicts, BTB misfetches, and instruction cache misses.*

in our simulation by a known CPI. For example, on the Pentium Pro, the CPI for *gcc* is around 1.4 [Bhandarkar & Ding 97]. Thus, the performance improvement we could expect with the global layout is approximately 15% (20 cycles saved[6] divided by 1.4). For the other benchmarks and reordering algorithms the improvements would be smaller but still non-negligible.

5 Conclusion

In this paper we have shown that code reordering algorithms improve the execution time of applications by the combination of reductions in instruction cache miss rates, branch mispredicts, and branch misfetches. We have quantified these reductions by implementing three variations of the Pettis and Hansen algorithm and simulating the reordered binaries of six SPEC95 benchmarks on two instruction set architectures: Intel IA32 and DEC Alpha.

Our simulations show that a *global* algorithm that allows one time in-lining of procedures and contiguity of basic blocks of different procedures is best at reducing the instruction cache miss rates, with a slightly better improvement for the IA32 instruction set architecture. The *global* layout and the two local layouts which retain the adjacency of blocks within the same procedure give similar improvements in branch prediction.

The relative degree to which applications will benefit from code reordering is more application specific than architecture-specific with those applications with a large code working set benefiting more.

As the depth of pipelines grows and the width of instruction issue widens, the penalties

[6]from Figure 10

for instruction issue slots lost because of instruction cache misses and of branch mispredicts will rise. Therefore mechanisms such as code reordering that will decrease the number of lost cycles due to these factors will increasingly become more important.

Future work could investigate reordering algorithms that consider correlation between the use of basic blocks, or some other form of temporal relation between the blocks, rather than a control flow graph based on total program execution frequencies. It would also be interesting to know whether the improvement in branch prediction accuracy brought upon by code duplication or by predication can compensate the potential harmful effects of code expansion on the instruction cache miss rate.

References

[Bhandarkar & Ding 97] Bhandarkar, D. and Ding, J. Performance Characterization of the Pentium Pro Processor. In *IEEE Micro*, pages 288–297, 1997.

[Bhandarkar 97] Bhandarkar, D. RISC versus CISC: A Tale of Two Chips. In *Computer Architecture News*, 25, pages 1-12, March 1 1997.

[Calder & Grunwald 94] Calder, B. and Grunwald, D. Reducing Branch Costs via Branch Alignment. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242-251, October 4-7, 1994.

[Digital Equipment Corporation 94] Digital Equipment Corporation. ATOM user manual, 1994.

[Fisher & Freudenberger 92] Fisher, J. and Freudenberger, S. Predicting Conditional Branch Directions From previous Runs of a Program In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85-95, October 1992.

[Hashemi, Kaeli, & Calder 97] Hashemi, A., Kaeli, D., and Calder, B. Efficient Procedure Mapping Using Cache Line Coloring In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[Hwu & Chang 89] Hwu, W. and Chang, P. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the ACM*, pages 242–251, 1989.

[Krall 94] Krall, A. Improving Semi-Static Branch Prediction by Code Replication In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 97-106, 1994.

[McFarling 89] McFarling, S. Program Optimization For Instruction Caches. In *Proc. 3rd International Conference On Architectural Support For Programming Languages And Operating Systems*, pages 183–191, April 1989.

[McFarling 92] McFarling, S. Combining Branch Predictors WRL Technical Note TN-36 June 1993.

[Microprocessor Report 95] P6 Underscores Intel's Lead vol 9, Number 2, Feb 1995.

[Pettis & Hansen 90] Pettis, K. and Hansen, R. Profile Guided Code Positioning. In *Proc. Conference On Programming Language Design And Implementation*, pages 16–26, June 1990.

[Romer, et. al. 97] Romer, T., Voelker, G., Lee, D., Wolman, A., Levy, H., Bershad, B., and Chen, B. Instrumentation and Optimization of Win32/Intel Executables. In Proc. USENIX Windows NT Workshop, August 11-13, 1997.

[Yeh & Patt 91] Yeh, T. and Patt, Y. Two-Level Adaptive Branch Prediction. In *Proceedings of the 24th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pages 51–61, November 1991.

[Young & Smith 94] Young, C. and Smith, M. Improving the Accuracy of Static branch prediction Using Branch Correlation In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232-231, October 4-7, 1994.