STRATEGIES FOR BRANCH TARGET BUFFERS

Brian K. Bray
M. J. Flynn

Technical Report No. CSL-TR-91-480

June 1991

Supported by NASA under NAG2-248 using facilities supplied under NAGW 419.

STRATEGIES FOR BRANCH TARGET BUFFERS

by

Brian K. Bray

M. J. Flynn

Technical Report No. CSL-TR-91-480

June 1991

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Abstract

Achieving high instruction issue rates depends on the ability to dynamically predict branches. We compare two schemes for dynamic branch prediction: a separate branch target buffer and an instruction cache based branch target buffer. For instruction caches of 4KB and greater, instruction cache based branch prediction performance is a strong function of line size, and a weak function of instruction cache size. An instruction cache based branch target buffer with a line size of 8 (or 4) instructions performs about as well as a separate branch target buffer structure which has 64 (or 256, respectively) entries.

Software can rearrange basic blocks in a procedure to reduce the number of taken branches, thus reducing the amount of branch prediction hardware needed. With software assistance, predicting all branches as not branching performs as well as a 4 entry branch target buffer without assistance, and a 4 entry branch target buffer with assistance performs as well as a 32 entry branch target buffer without assistance. The instruction cache based branch target buffer also benefits from the software, but only for line sizes of more than 4 instructions.

Key Words and Phrases: branch target buffer, instruction cache

Contents

1 Introduction

Concurrent execution of multiple instructions in the same processor is used to speed the execution of programs. Instructions begin execution before the previous instructions have completed. Since future instructions are fetched before the current instructions have completed, branch instructions can hurt the performance of such machines. Branches change the location from which the next instructions is fetched, and the actual location is not available until the branch has completed. To address this problem in pipelined machines, the location of the future instructions must be predicted whenever a branch is encountered. Therefore, achieving high instruction issue rates can depend on the ability to dynamically predict branches.

2 Benchmark Branch Characterization

Before investigating schemes to improve branch prediction performance, we observed the branching characteristics of our applications (see Appendix). One concern is the dynamic frequency of branches. Figure 1 shows that for supercomputer-type core routines (fast fourier transform, and Livermore Loops), branches are a small part of the dynamic instruction stream. However, for the workstation-type applications (compress, espresso, gnu C compiler, spice3, TeX, etc.), branches average a non-trivial 22 percent of the instructions executed. Figure 2 shows the percent of branches taken. The branches in the supercomputer-type core routines practically always branch. For the workstation-type applications, branches are taken an average of 68 percent of the time. The supercomputer-type core routines have few branches, and the branches are very predictable in that they almost always are taken. This makes very small and simple hardware schemes look very good, therefore the supercomputer-type core routines (fft, lloops) will not be considered in any of the following results.

The branch type distribution for the workstation-type applications is shown in Figure 3. The branch distribution is dominated by conditional branches, 57 percent of which are taken, followed by the unconditional branches for which the target is known (direct jump, direct call), and then unconditional branches for which the target is unknown (return, indirect jump, indirect call).

The conditional branches can be further broken down into forward and backward conditional branches, Figure 4. Forward conditional branches are 67 percent of the conditional branches, and 48 percent of them are taken. On average backward conditional branches are 33 percent of the conditional branches, and 77 percent of them are taken.

The conditional branches (Figures 5 and 6) are weighted according to number of times executed, and they are separated into groups depending on their individual branch taken percentage (the number of times they branched divided by the number of times they were executed). Notice that most of the branches executed have a tendency to have a very low or very high branch taken percentage. It is useful to know that statistically branches have a tendency to either frequently branch or rarely branch. Hardware and/or software can be used to take advantage of this tendency. One interesting item to note in Figure 6 is that there is surprising number of backwards conditional branches with a low branch taken percentage. These branches are mainly due to the closure branch at the end of some *while* loops. These particular *while* loops rarely iterate more than once and the first iteration through the *while* loop is protected by a forward conditional branch.

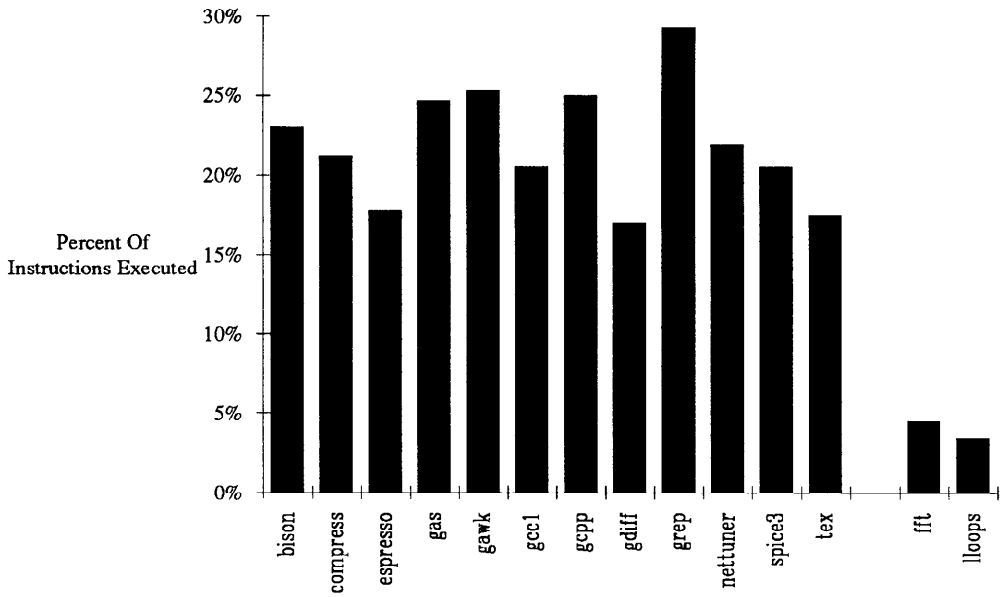

Figure 1: Branch Frequency

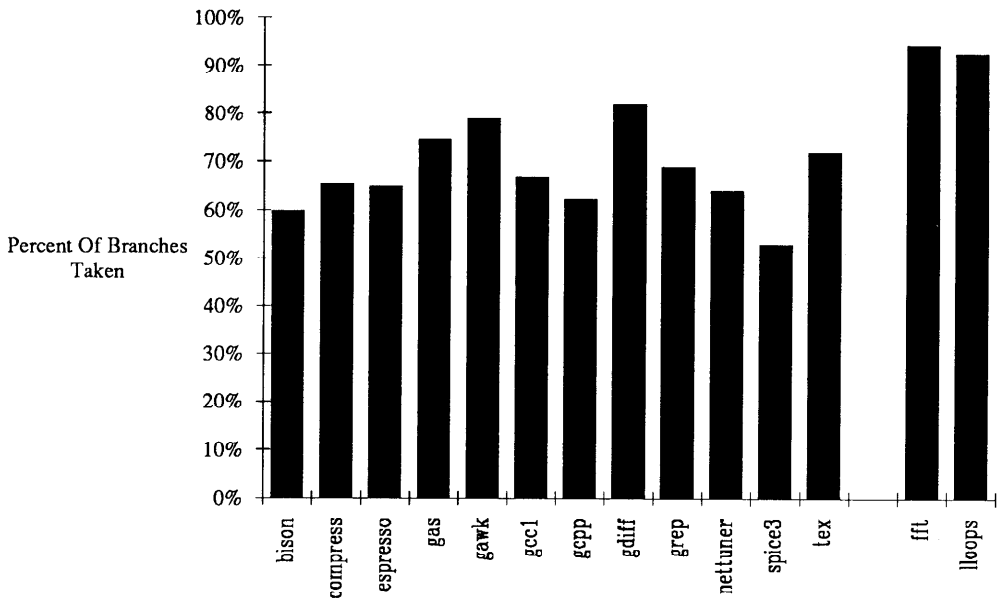

Figure 2: Branches Taken

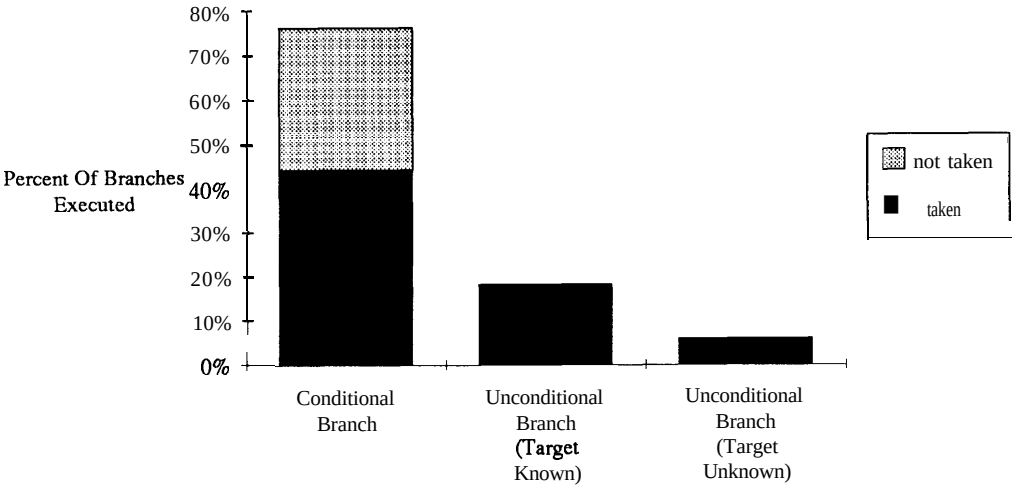

Figure 3: Branch Type Distribution

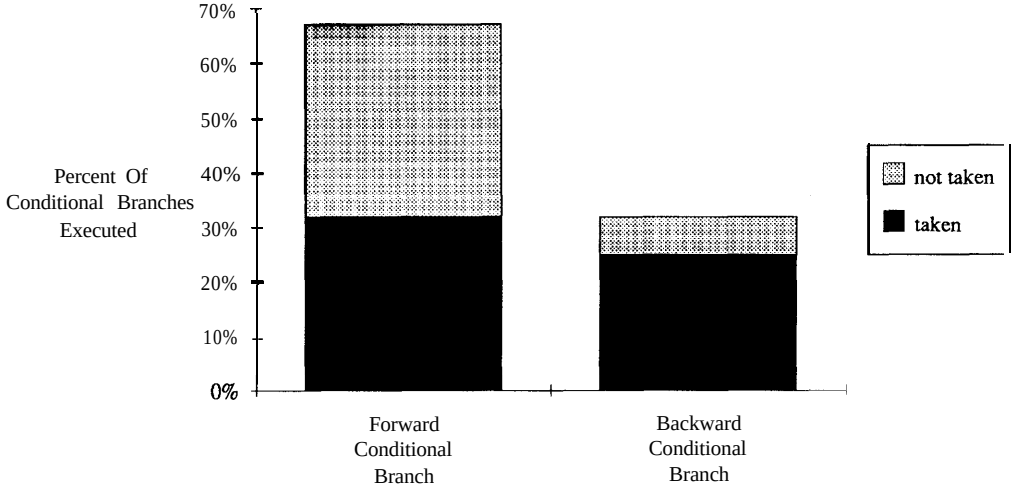

Figure 4: Conditional Branch Type Distribution

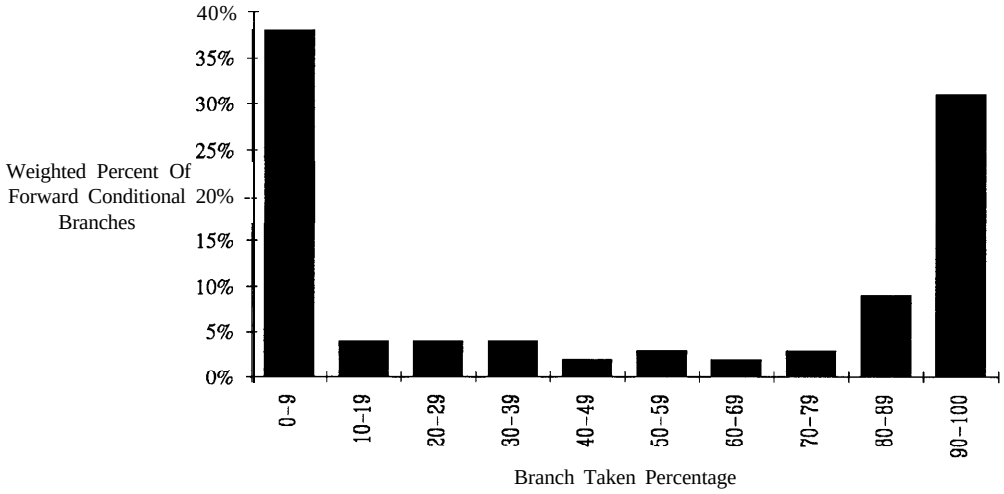

Figure 5: Forward Conditional Branch Taken Distribution

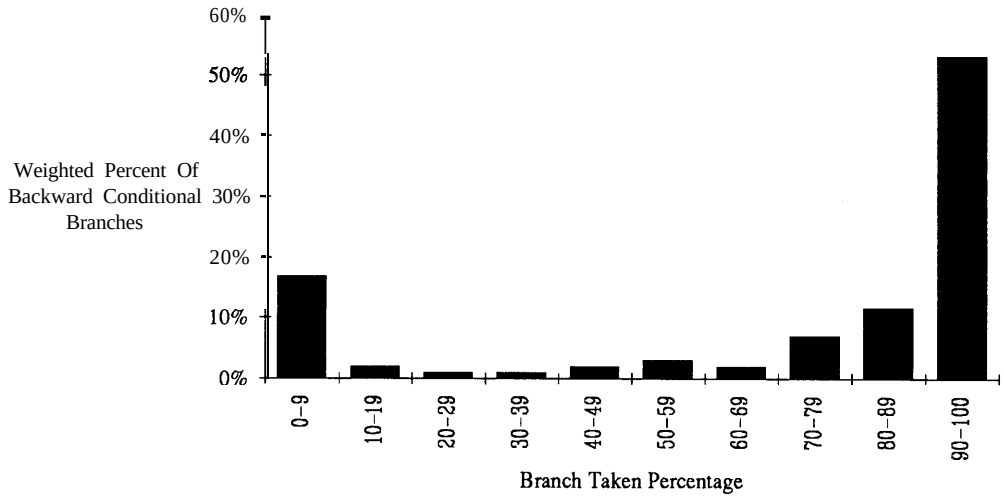

Figure 6: Backward Conditional Branch Taken Distribution

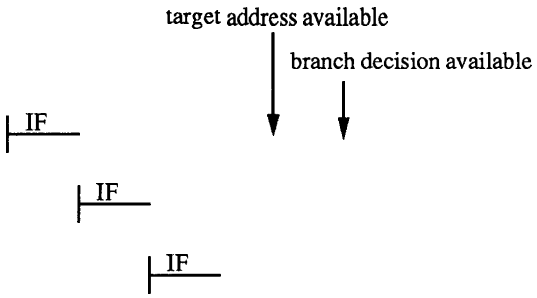

Figure 7: Why The Target Can Not Be Guessed

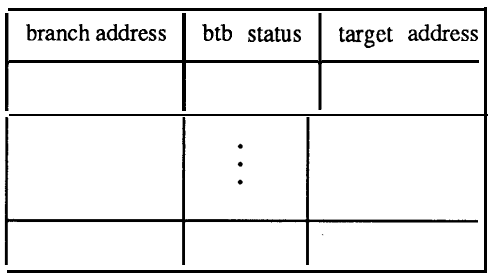

Figure 8: Separate Branch Target Buffer Structure

3 Branch Target Buffer Design Alternatives

Since backward conditional branches usually go to the target and unconditional branches always go to the target, the default for those branches should be to guess the target address. However, to avoid pipeline delays the address of the next instruction may be needed before the instruction is decoded and the target address computed (Figure 7). Consequently, some mechanism is needed to predict the branch target address by just using the address of the branch instruction being fetched. One hardware method to predict branch targets is to use branch target buffers (btb) [LS84]. BTBs act as one entry per line caches where the index is the address of the branch instruction and the target address is the cached value.

One way to implement btbs is to have a separate branch target buffer structure (Figure 8). Another implementation is to have the branch target buffer structure incorporated in the instruction cache [Joh89] (Figure 9). The instruction cache based btb has advantages and disadvantages in that the address tag is shared with the instructions. The advantages are that there is no need for a duplicate tag set, and that branch prediction hardware can be easily added since it is an extension of the instruction cache. The main disadvantage is that there can only be one branch target per unit of instructions, hence closely spaced branches could have contention for the single branch target entry. Another disadvantage is that there could be contention for tag access between instruction fetching and the branch predictor modifying the branch target address or prediction information.

Past attention in btb design focused on hit rate to describe the performance, but hit rate is not all that important. How often the instruction fetch unit predicts the correct address is the important performance

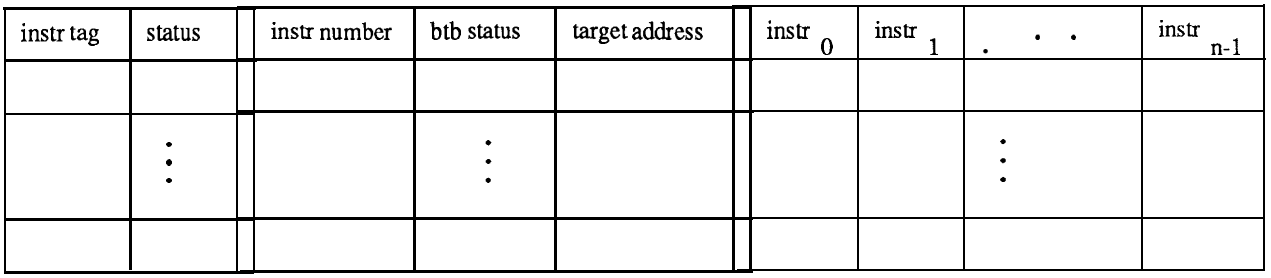

Figure 9: Instruction Cache Based Branch Target Buffer Structure

issue. A branch can miss in the btb and still be predicted correctly, since the default is to go inline. Because miss rate does not accurately show the performance of the btb, *we* use *predict incorrectly* as a measure of performance.

Figure 10 shows the effects of the workload. The dynamic branch and predicted branch taken frequencies of the supercomputer-type applications indicate that it is straightforward for a small btb to get good performance. For this reason we do not include the supercomputer-type applications in the following results.

Figure 10 describes various btb structures as a function of the number of btb entries. The origin is *stop,* which signifies that there are no btb entries and that the instruction fetch predictor stops when it comes to a branch, thus always predicting incorrectly. The next strategy is *inline*, which means that there are no btb entries, but the instruction fetch predictor guesses the target is not taken when it comes to a branch. Other strategies specify the number of btb entries. If the branch address is cached in the btb structure then the instruction fetch will predict the path pointed to by the associated cached value. If there is no corresponding entry the instruction fetch predictor guesses inline. (Note that 100% – *inline* equals the performance if the target were always predicted. As explained earlier, latency prevents this from being an option.)

All btb organizations are direct-mapped unless otherwise stated. The default branch target buffer organization is direct-mapped for the same reason as caches are direct-mapped [Hil88]: low access time. An entry is allocated to the btb when a branch is taken. An entry is deallocated when the prediction is incorrect or the entry collides with the new entry that is being allocated.

For the separate btb organization, the number of entries is very important. The number of entries in an instruction cache based btb organization depends on how many instructions are associated per btb entry and the size of the instruction cache. In Figure 11, as the size of the direct-mapped instruction cache increases from 1KB to infinity, the branch prediction rate improves only marginally for instruction caches larger *than* 4KB *(instructionsixe* = 4B). However, as the numbcr of instructions associated per btb entry decreases, the prediction rate improves noticeably.

The branch prediction performance improves for shorter line sizes, not because there are more btb entries but because there are less spatially local instructions to share an entry. (An infinite sized i-cache with 16 instructions per btb entry is easily outperformed by a 4KB i-cache with 4 instructions per entry.) Figure 12 shows the performance comparision of separate btbs verses an infinite instruction cache based btb structure. A 16 entry separate btb performs better than the i-cache based btb structure when the line size

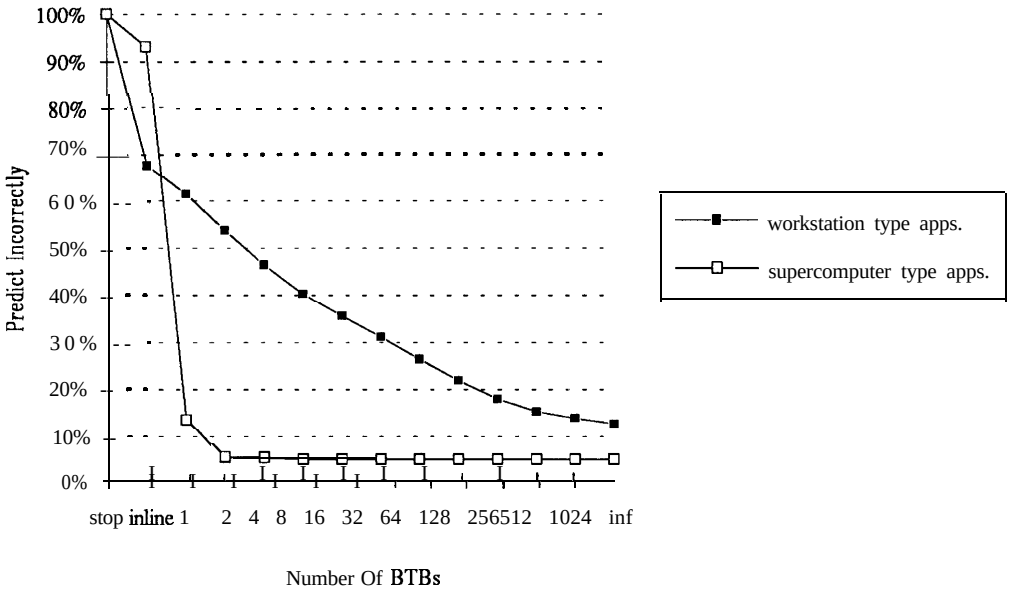

Figure 10: Separate BTB Structure Performance For Different Workloads

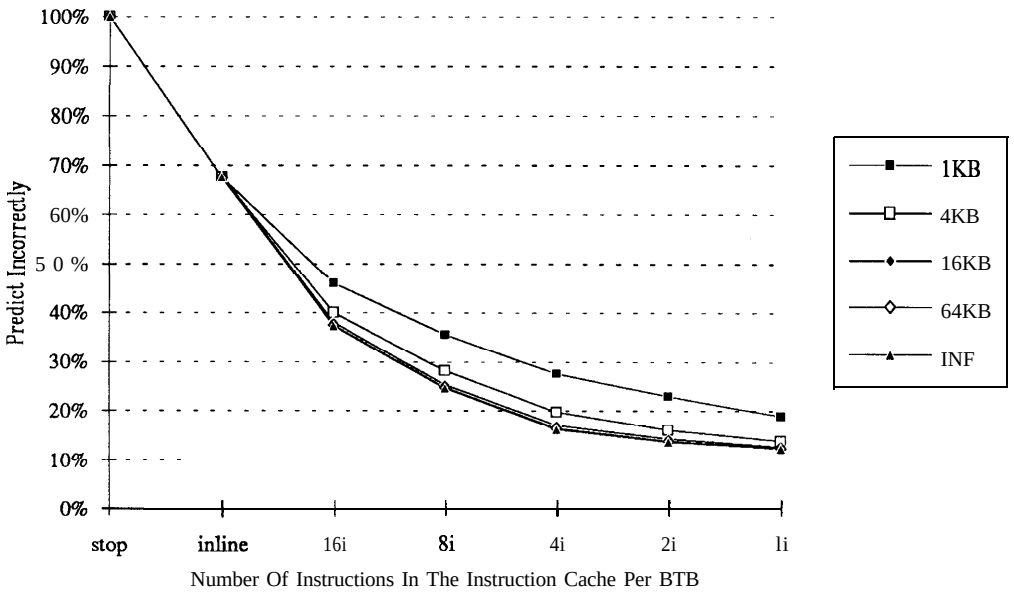

Figure 11: Effect Of Instruction Cache Size On I-Cache Based BTBs

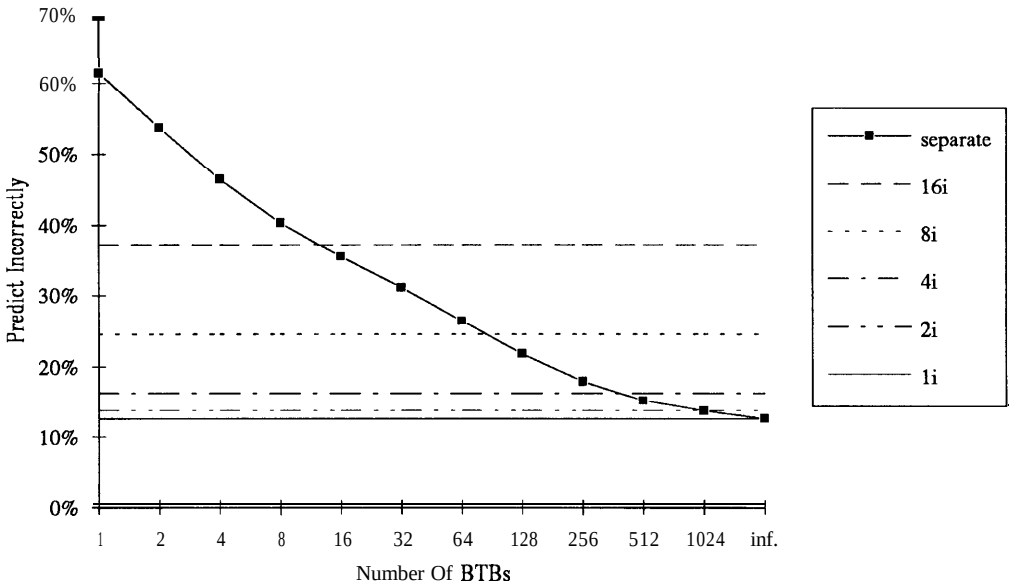

Figure 12: Performance of Separate vs. I-Cache Based BTBs

is 16 instructions, even though the i-cache based btb structure has many more btb entries.

A large number of btb entries is useful when the instruction stream returns to a program locality but the granularity of the btb entry sharing is important for when computation is localized in a section of the program address space. (The i-cache fetch size can be larger than the line size.)

There is little benefit in reducing the line size to less than 4 instructions, since it is unlikely that there are multiple taken branches in the line. There is a slight improvement in decreasing the line size from 2 instructions to 1. This is because we simulated the assembly code before the assembler filled delay slots with non-branch instructions or no-ops.

Since the number of instructions per btb entry is important and there is little btb performace difference for instruction cache sizes of 4KB and greater, all of the following i-cache based btb performance results assume an infinite size instruction cache.

3.1 Associativity

Figures 13 and 14 show the effects of increasing the associativity from the direct-mapped (a=l) case. Associativity can improve performance, because entries are less likely to be removed because of contention. At sizes of 2 to 4 btb entries, the more associative organizations perform better than the direct-mapped equivalents because they more quickly capture the branches of small to medium size loops (there is less collisions for branch entries within a loop). At sizes of 16 to 128, the more associative organizations more quickly capture the branches in the larger loops and the frequently called functions. However, associativity may slow the btb access down to the extent that the btb could become the critical path. From Figure 13, for sizes up to and including 16 entries, doubling the number of entries gives more performance than increasing the associativity. For sizes from 32 to 64 entries, doubling the number of entries approximately equals the effect of increasing the associativity. It is only for 128 and 256 entries that 4-way and full associativity outperform doubling the number of direct-mapped entries. Associativity has little effect on i-cache based btb organizations, since direct-mapped instruction caches of 4KB and

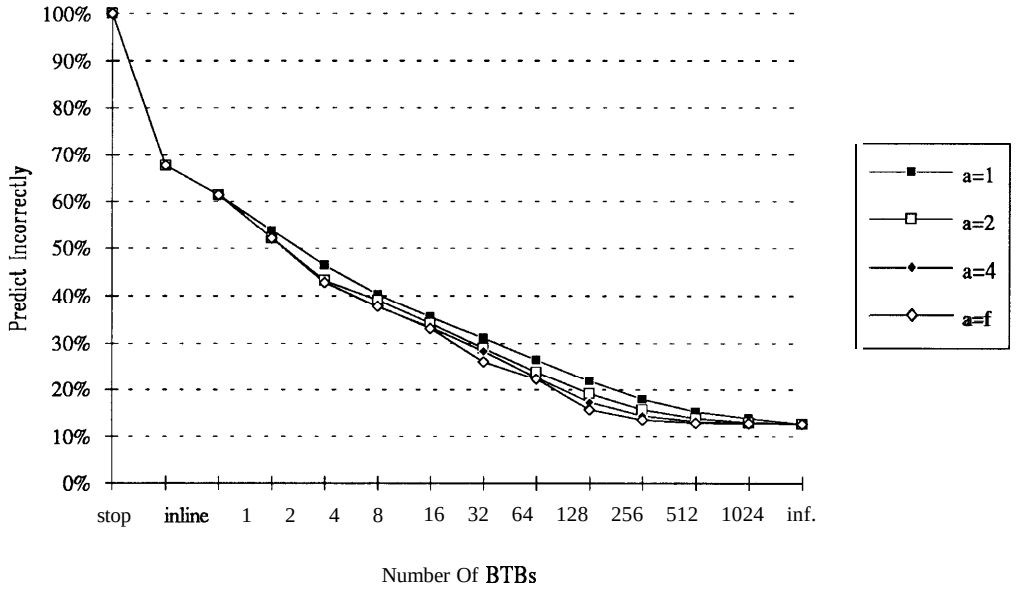

Figure 13: Effect Of Associativity On BTB Performance

larger have btb performance almost equivalent to that of an infinite cache.

3.2 Effects of History and Inline Allocation

Thus far in this study, we have been using a prediction based only the last branch decision (h=1) to determine whether the branch should be cached, and we have only allowed taken branches to be cached. [Smi8 1] and [LS84] have shown that increasing the history state can improve prediction performance. In Figures 15, 16, 17 and 18, we see the effects of increasing the history state and allowing branches which are not taken to be allocated. A history state of 2 (h=2) means that the branch can be mispredicted twice before being deallocated. Inline allocate means that in addition to allocating branches which are taken, branches which go inline are also allocated.

The relative performance graphs (Figure 16 and 18) are similar in shape. Even though the btb structures are different for these two figures, the same effects occur. When the history state is increased from 1 to 2, the relative prediction performance improves as the number of btb entries is increased. As the number of btb entries is increased, it is less likely that a btb entry is replaced because of capacity limitations, therefore it is more likely a cached branch is reaccessed. The increased history state makes it harder for a different branch result to cause the entry to change its prediction. Recall that in Figures 5 and 6 we have seen that an individual conditional branch has a strong tendency to branch or not to branch.

As expected, when the history state is 1 and inline branches are allowed to be allocated, performance is reduced due to btb cache pollution. Branches which go inline push out entries of branches which had recently taken, and since the default is to go inline there is no benefit to predict inline. Not until the number of btb entries is infinite (or 1 per instruction, li) does the performance equal that of not allowing inline allocation. *The hump* in the relative performance graphs is the result of the h=l model capturing locality earlier than the h=l with inline allocation model. From this reasoning, it seems that branches which go inline should never be allocated. However, if the history state is greater than 1, allocating a branch that goes inline could prevent a branch that rarely branches from getting allocated as a branch which branches. Even with increased history state, though, btb cache pollution caused by inline allocation

Figure 14: Relative Effect Of Associativity On BTB Performance

dominates unless the there are a large number of btb entries (or very low number of instructions per btb entry).

As mentioned earlier, i-cache based btb structures need to access the i-tag to update the btb entry. Therefore, there are more i-tag accesses than instruction accesses. These extra i-tag accesses may cause contention with the instruction fetch unit. The extra i-tag accesses needed are shown in Figure 19. As the number of instructions per btb entry decrease, the contention for the btb entry decreases so the extra i-tag accesses decrease.

3.3 Effect Of Limiting Deallocation of Backward Conditional Branch Entries

If the history state is 1, when a loop finishes its last iteration, the backwards conditional branch fails to branch and goes inline causing it to be deallocated from the btb cache. If the history state were greater than 1, most likely a backward conditional branch would be wrong the very first time it was encountered and everytime the loop ended, compared to everytime the loop started and ended (iterations > 1).

By not deallocating backward conditional branch entries when they go inline, we hoped to get some of the benefit of extra history state without the overhead. The result was a very slight prediction improvement, but the improvement was limited because of the backward conditional branches that rarely branch. Once allocated they mispredicted, negating most of the benefit of the backward conditional branches that usually go to the target. In Figures *20,* however, we see that not deallocating backward conditional branches reduces the number of extra instruction tag accesses that must be made for the i-cache based btb.

3.4 Effect Of Not Allocating Unconditional Branches Whose Target Is Unknown

Unconditional branches whose target is unknown at compile time include *returns, indirect jumps, and indirect calls.* They comprise on average 6% of the branches executed. Initially it was thought that not allowing such branches to be allocated might improve performance for the btbs with fewer entries

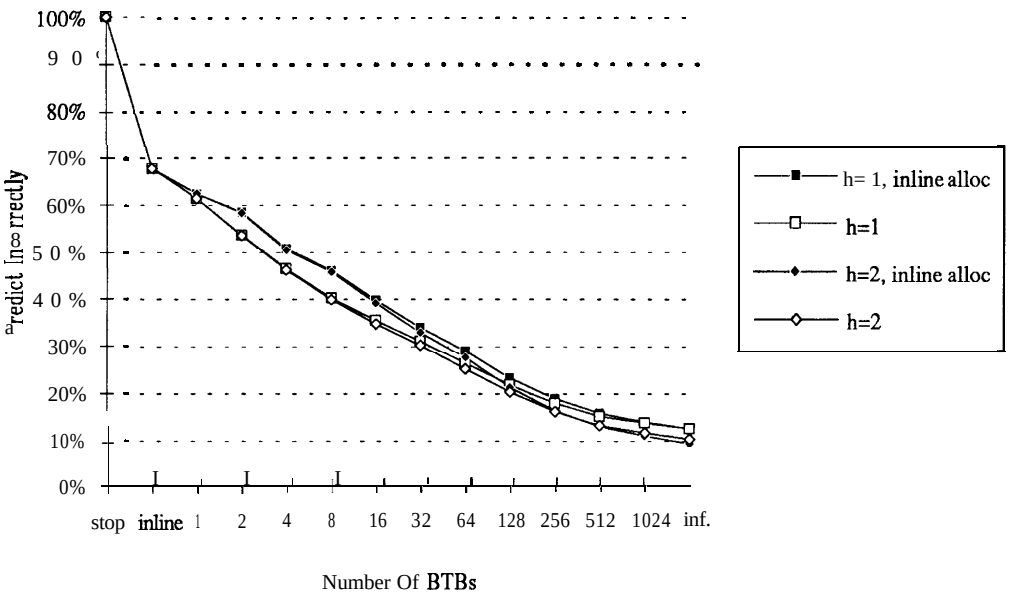

Figure 15: Effect Of History and Inline Allocation On BTB Performance

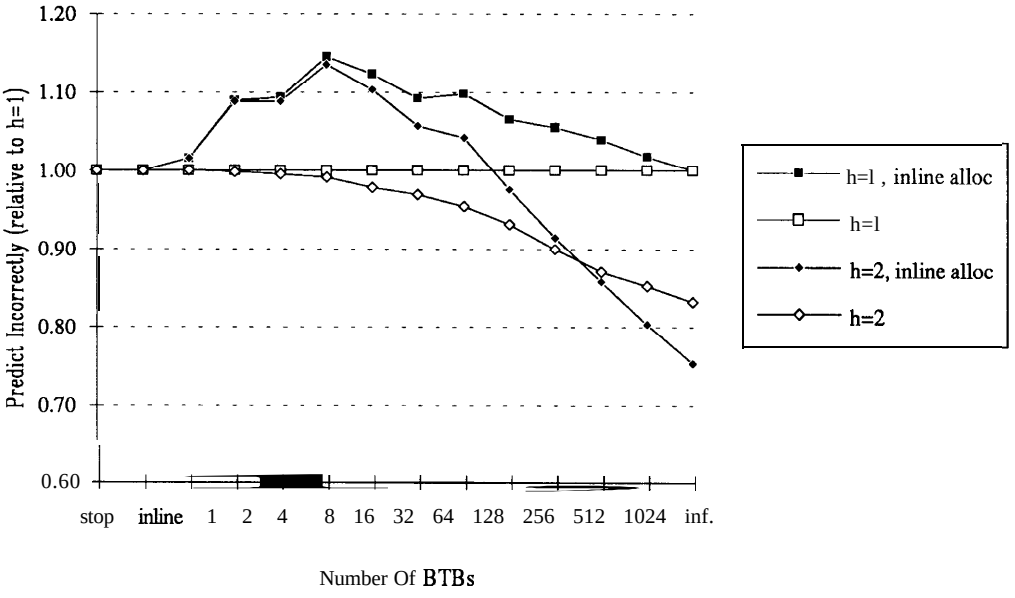

Figure 16: Relative Effect Of History and Inline Allocation On BTB Performance

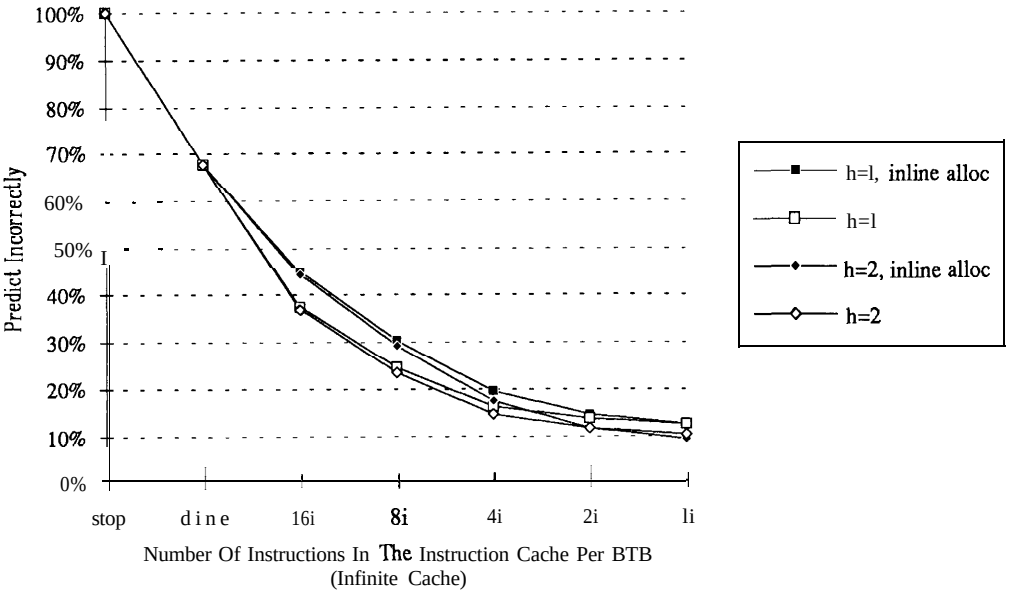

Figure 17: Effect Of History and Inline Allocation On I-Cache Based BTB Performance

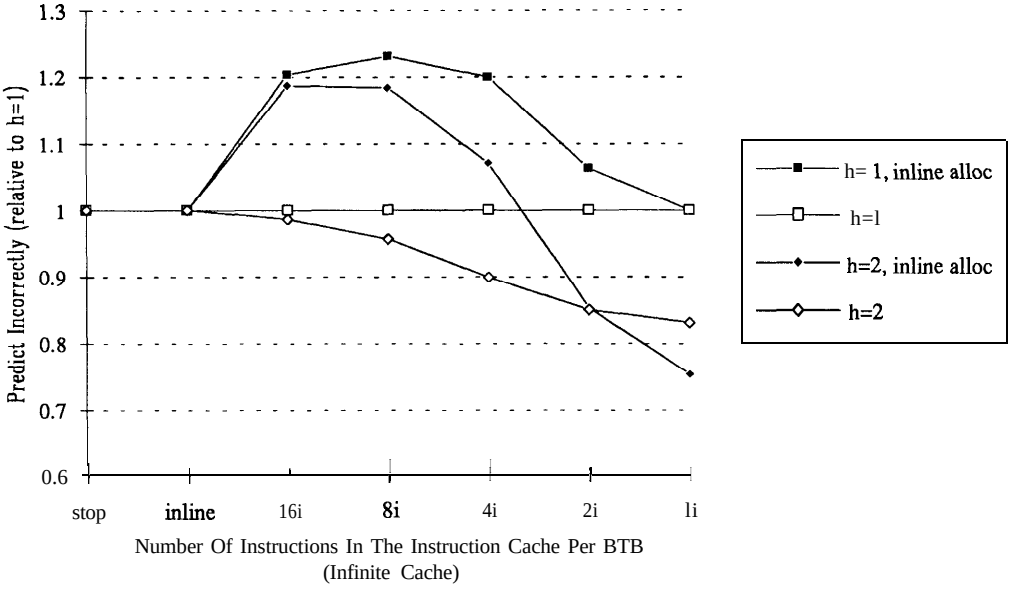

Figure 18: Relative Effect Of History and Inline Allocation On I-Cache Based BTB Performance

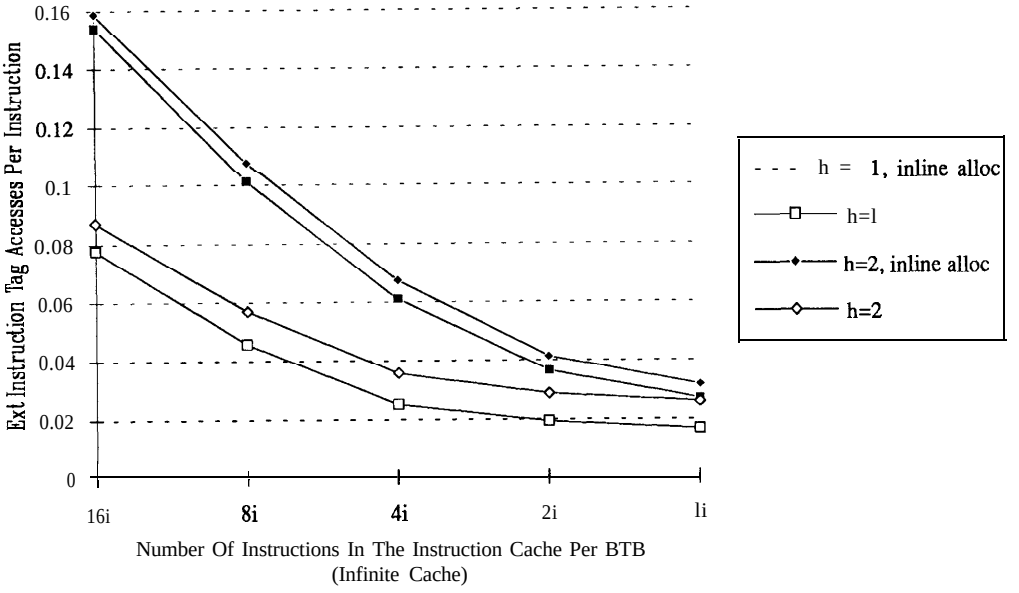

Figure 19: Extra I-Tag Requests for I-Cache Based BTBs

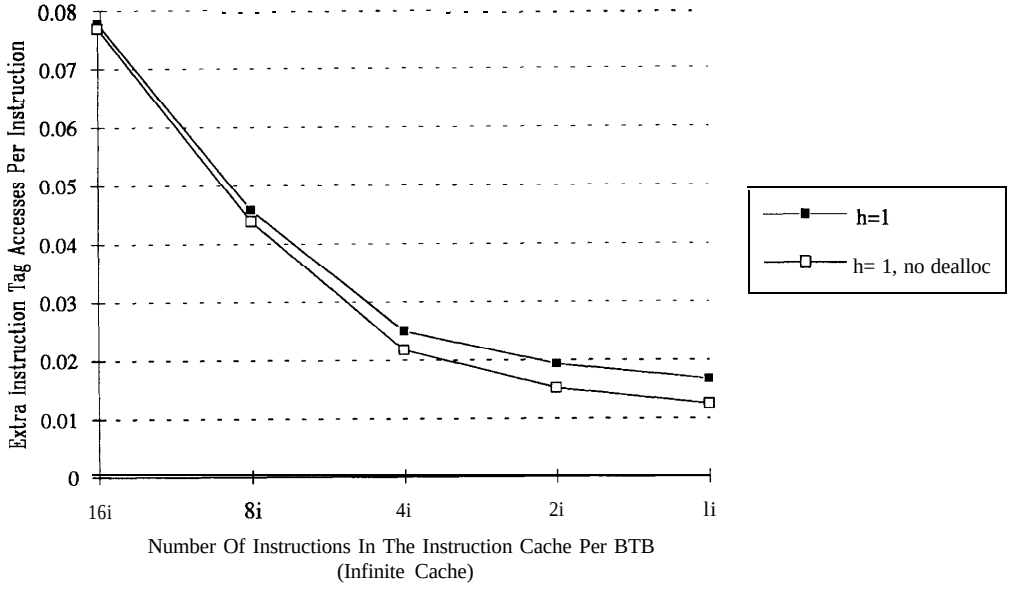

Figure 20: Effect Of Not Deallocating Backward Conditional Branches Even Though There Was Miss Prediction

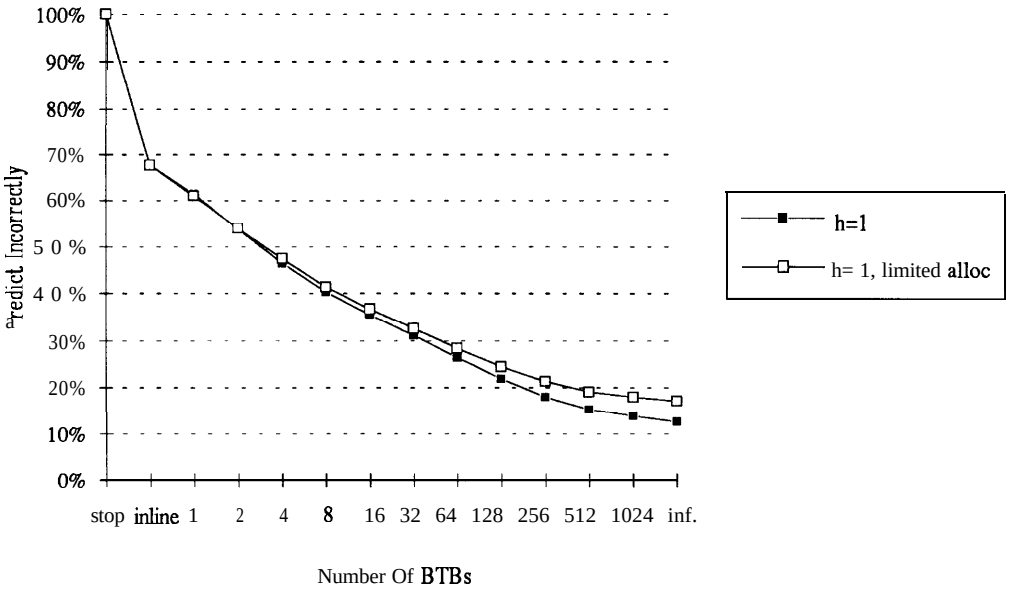

Figure 21: Effect Of Not Allowing Branches With Unknown Targets To Be Allocated

by preventing cache pollution. However, we found that is not true because most of the unconditional branches whose target is unknown at compile time are *returns*. In the case of *returns*, the working set is changing, therefore allocating for small btb organizations does not cause btb cache pollution. Also, we initially thought that the variable target nature of these branches would prevent btbs from providing little benefit, however this is not so. Even though unconditional unknown target branches comprise an average of 6% of the branches, preventing them from being allocated caused on average a 4% loss in prediction for the infinite btb case (Figures 21 and 22). Most of the unconditional unknown target branches were *returns* and that most procedures/functions are usually called from only one place. The only benefit of not allocating the unconditional unknown target branches is that it reduces the number of extra instruction tag accesses that the i-cache based btb has to make (Figure 23). The reduction in i-tag accesses does not seem worth the loss in prediction ability.

3.5 Effect Of Using Software InterBlock Reorganization To Reduce The Number Of Forward Conditional Branches That Are Taken

Most forward conditional branches were found to have a tendency to branch rarely or frequently (Figure 5). This allows the branch prediction hardware to determine what to predict when the branch is encountered later in the execution. By taking profiling information about the forward conditional branches, software can perform interblock reorganization within a procedure, such that the most likely path of execution for forward conditional branches is to go inline. This is similiar to trace scheduling [Fis81] in that the mostly likely path of execution is kept in the main path, and unlikely traces are moved out of the way. If a program is rearranged such that it is more likely that a forward conditional branch goes inline, fewer btb entries are needed to achieve the same level of performance.

Using perfect profiling information, we reorganized the basic blocks in the procedures of our applications/benchmarks to reduce the number of forward conditional branches which are taken. For *if-then-else* structures, *we* moved *the then* or else blocks out of the main trace depending which was used least. For *if-then* structures, *we* moved the *then* blocks out of the main trace if the profiling information indicated

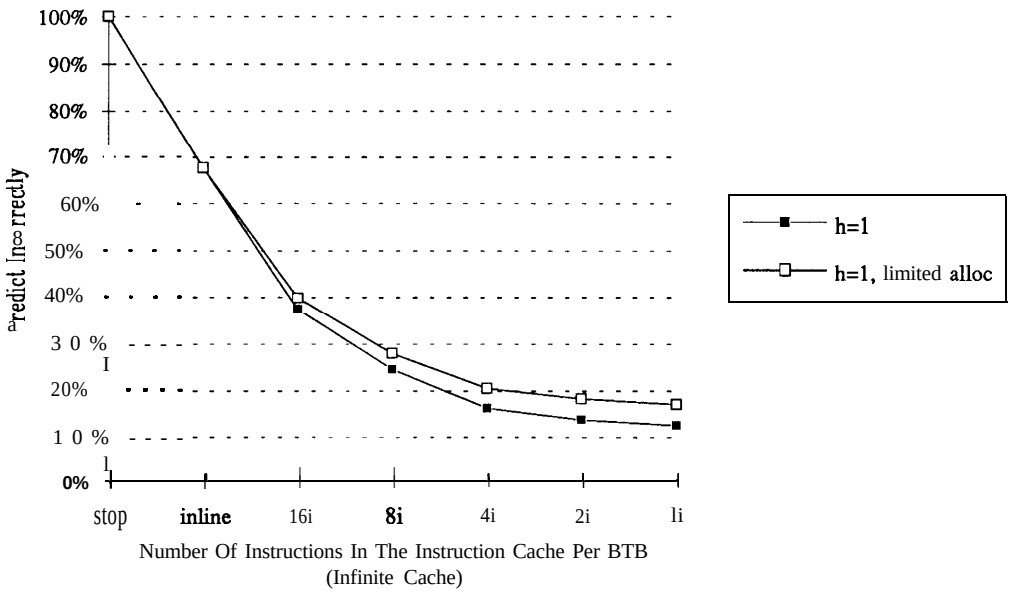

Figure 22: Effect Of Not Allowing Branches With Unknown Targets To Be Allocated

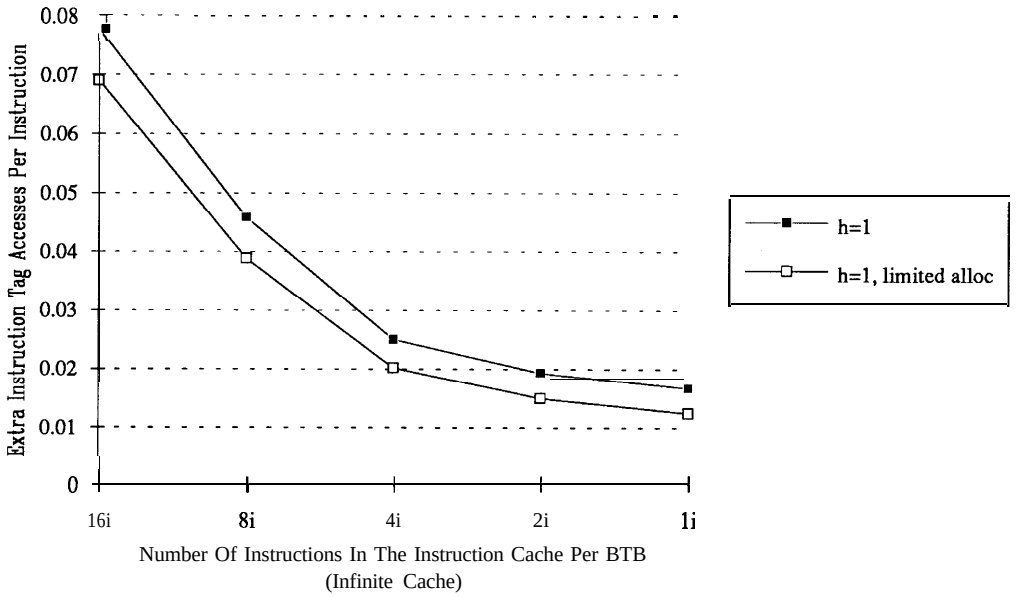

Figure 23: Effect Of Not Allowing Branches With Unknown Targets To Be Allocated

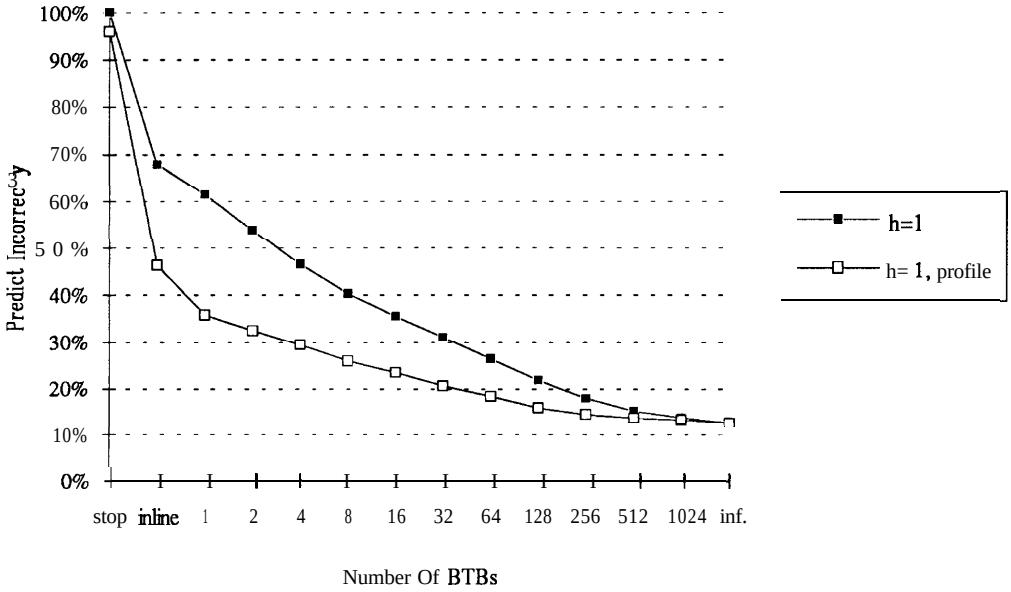

Figure 24: Effect Of InterBlock Reorganization

there was an 80% or better chance that the *then* structure is skipped. The 80% figure was chosen (vs a 50% figure) because moving a *then* structure out of the main trace entails adding an extra unconditional branch instruction for *the then* clause to get back to the main trace. The unconditional branch associated with the *then* clause of an *if-then-else* is transfered to the clause that is moved out of the main trace. Because the unconditional branch is moved out of the main trace in the *if-then-else* clauses, the programs execute fewer branches so even the *stop* category is improved by this interblock reorganization (Figures *24* and 25).

For our benchmarks/applications this interblock reorganization can make organizations which always predict inline perform as if they have 4 btbs, or a 4 btb machine perform as if it has 32 btbs (Figures *24).*

As the number of btbs increase (or instructions per btb decrease), the performance benefit of this interblock reorganization decreases. The software and hardware take advantage of the same effect, so as the amount of hardware approaches infinity, the performance benefit becomes negligible.

In Figure 26 the interblock reorganization significantly reduces the extra i-tag accesses per instruction for the longer line sizes, because there are on average less taken branches within a line. The number of extra i-tag accesses per instruction slightly increases for the smaller line sizes. Some of the increase is due the fact that the total number of instructions executed had been decreased by moving unlikely else clauses out of the main trace, thus elminating the unconditional branch at the end of the likely *then* clause. The rest of the increase is due to encountering more unique branches: When an unlikely *then* clause of an *if-then* structure is moved out of the main trace, an unconditional branch is added to the *then* clause for it to return to the main trace should it be taken. If the unlikely *then* clause is taken, then an extra unique branch is executed and allocated thus increasing the number of i-tag accesses.

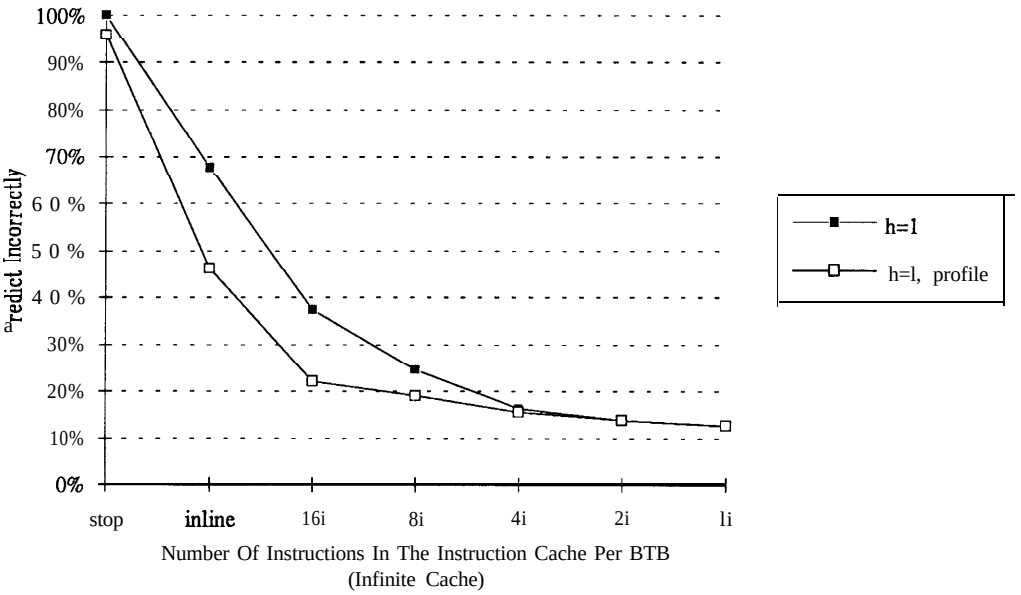

Figure 25: Effect Of InterBlock Reorganization

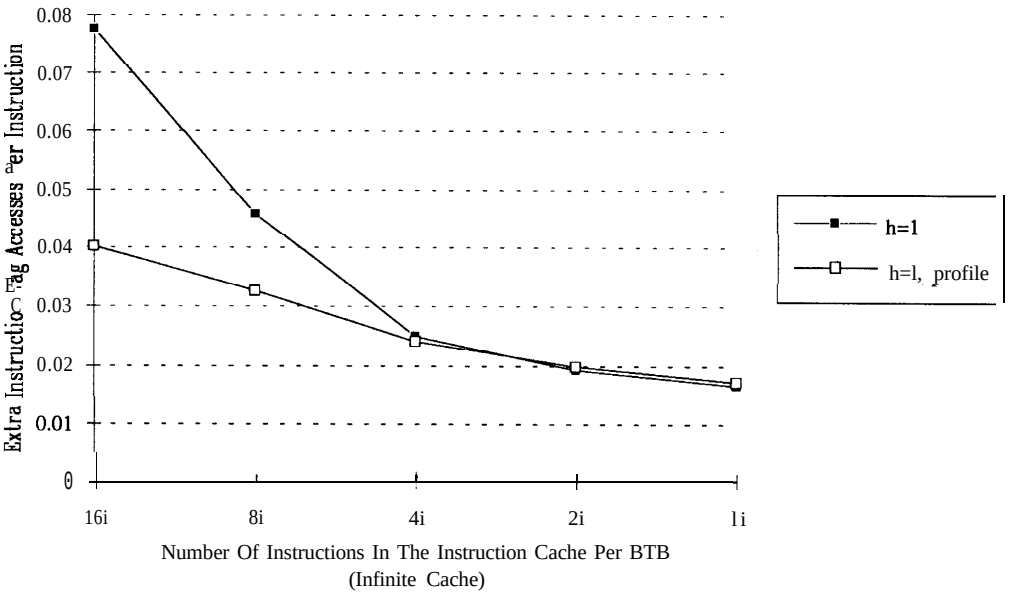

Figure 26: Effect Of InterBlock Reorganization

4 Conclusion

For instruction caches of 4KB and greater, instruction cache based branch prediction performance is a strong function of line size, and a weak function of instruction cache size. An instruction cache based branch target buffer with a line size of 8 (or 4) instructions can perform about as well as a separate branch target buffer structure which has 64 (or 256, respectively) entries. Simple direct-mapped branch target buffer structures, which require no extra history state and only allocate taken branches, perform almost as well as more complex organizations. Not deallocating backward conditional branches when they do not branch to the target provides little prediction improvement but does reduce the number of extra instruction tag accesses caused by an instruction cache based branch target buffer.

Software interblock reorganizing for reducing the number of taken forward conditional branches provides little benefit to schemes where the amount of branch hardware approaches infinity, but it significantly improves the performance of small btb organizations. With software assistance, predicting all branches as not branching performs as well as a 4 entry branch target buffer without assistance, and a 4 entry branch target buffer with assistance performs as well as a 32 entry branch target buffer without assistance. The instruction cache based branch target buffer also benefits from the software, but only for line sizes of more than 4 instructions.

References

[Fis81] J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions Computer, C-30(7):478-490,* July 198 1.

[Hil88] Mark Hill. The Case for Direct-Mapped Caches. *IEEE Computer, 2* 1(12):25–40, December 1988.

[Joh89] William Johnson. Super-Scalar Processor Design. Technical Report No. CSL-TR-89-383, Computer Systems Laboratory, Stanford University, June 1989.

[LS84] J. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer,* 17(1):6–22, January 1984.

[Smi81] James E. Smith. A Study Of Branch Prediction Strategies. In *Conference Proceedings, The 8th Annual Symposium on Computer Architecture,* pages 135-148, 1981.

A Appendix:

A.1 Benchmarks

Trace driven simulation produced the presented results. The architecture simulated was essentially the MIPS R2000/R3000. Note that the R2000/R3000 does not have 64-bit floating point loads and stores, a 64-bit load or store is made by using two 32-bit transfers. However, the model we use here of the MIPS R2000/R3000 has 64-bit floating point loads and stores. The following C benchmarks were optimized with the Ultrix 4.0 C compiler with optimization level 02. The code executed in the application code, string routines, and printf routines was simulated, while the code executed in system calls, scanf routines, and math libraries was not.

Name	Description	Instr. (10^6)	% loads	% stores	% branches
`fft`	fast fourier transform - 1024x1024 2-D fft	7.93	31.7	16.1	4.5
`lloops`	composite of the 14 Livermore Loops	0.0645	30.8	12.4	3.4

Table 1: Instruction Distribution of Supercomputer Type Applications

Name	% forward conditional branches	% backward conditional branches	% unconditional branches (target known)	% unconditional branches (target unknown)	% branches taken
`fft`	0.1	98.7	0.6	0.6	94.0
`lloops`	6.8	85.0	7.5	0.7	92.3

Table 2: Branch Distribution of Supercomputer Type Applications

Name	Description	Instr. (10^6)	% loads	% stores	% branches
bison	gnu yacc (LALR parser generator) - parsing cexp.y of the gnu C compiler	4.76	28.2	16.4	23.0
compress	reduces the size of a file using adaptive Lempel-Ziv coding - compressing a 150KB tar file	12.3	19.8	10.2	21.2
espresso	boolean expression minimizer - reducing a 14-bit input, 8-bit output PLA matrix	150	21.1	4.5	17.7
gas	gnu assembler - assembling a 1800 line file	6.54	23.0	13.3	24.7
gawk	gnu awk - patern matching and processing on a 304 line file	1.42	26.7	14.1	25.3
gcc1	gnu C compiler version 1.36 - compiling (and optimizing) to assembly code a 1500 line C program	43.6	23.4	13.1	20.5
gcpp	gnu C preprocessor - preprocessing a 1500 line C program	0.997	17.8	9.4	25.0
gdiff	gnu diff - comparing two 1112 line files	1.13	23.7	13.0	17.0
grep	grep - search file for regular expression	0.191	18.5	13.3	29.3
nettuner	nettuner - reads a netlist of a 4x4 multiplier and pads the circuit delays	12.3	21.2	14.7	21.9
spice3	circuit simulator - simulation of a Schottky TTL edge-triggered register	149	37.7	9.1	20.5
tex	document preparation system - formatting of a 14 page technical report	82.0	22.6	15.9	17.4

Table 3: Instruction Distribution of Workstation Type Applications

Name	% forward conditional branches	% backward conditional branches	% unconditional branches (target known)	% unconditional branches (target unknown)	% branches taken
bison	52.8	20.8	23.5	2.9	59.9
compress	60.1	21.3	17.0	1.6	65.5
espresso	55.5	30.4	9.7	4.4	64.8
gas	52.0	19.7	19.7	8.6	74.6
gawk	52.1	13.0	22.6	12.3	78.9
gcc1	59.3	17.4	15.4	7.9	66.8
gcpp	39.2	23.2	22.3	15.3	62.1
gdiff	30.2	61.5	6.1	2.2	81.8
grep	56.3	27.7	15.2	0.9	68.9
nettuner	52.1	16.9	24.3	6.8	63.8
spice3	52.9	36.4	8.8	1.9	52.9
tex	50.6	10.1	27.7	11.6	71.7

Table 4: Branch Distribution of Workstation Type Applications