Published in *Proceedings of the International Parallel Processing Symposium, April 1994, Cancun, Mexico.*
pp. 172-179

# The Effect of Speculative Execution on Cache Performance

Jim Pierce[1] and Trevor Mudge

University of Michigan

1. Jim Pierce was supported in this work by a grant from the Intel Corp.

## Abstract

*Superscalar microprocessors obtain high performance by exploiting parallelism at the instruction level. To effectively use the instruction-level parallelism found in general purpose, non-numeric code, future processors will need to speculatively execute far beyond instruction fetch limiting conditional branches. One result of this deep speculation is an increase in the number of instruction and data memory references due to the execution of mispredicted paths. Using a tool we developed to generate speculative traces from Intel architecture Unix binaries, we examine the differences in cache performance between speculative and non-speculative execution models. The results pertaining to increased memory traffic, mispredicted path reference effects, allocation strategies, and speculative write buffers are discussed.*

## 1 Introduction

Parallel computation is an increasingly important research area in the quest for faster general purpose microprocessors. Superscalar processor architectures exploit instruction-level parallelism to concurrently run multiple instructions per cycle. Some current implementations can issue two instructions per cycle and several can issue four or even six instructions at once under certain conditions [1][3][4]. Research has shown that more instruction-level parallelism exists in often used, non-numeric code but architecture designers face many problems in harnessing this potential concurrency. One problem is that the length of an unbroken stream of instructions, a basic block, averages from 3.0 - 6.5 instructions in applications such as those found in the SPEC benchmark suite. Thus to achieve high issue rates instructions must be fetched the basic block ending conditional branches. This can be accomplished by speculatively executing instructions past branches, i.e., the direction of the branch is guessed before the branch condition is computed in the pipeline and execution continues in the predicted direction until the branch is resolved. If the prediction were incorrect, the processor state must be restored to the state prior to the predicted branch and restart execution down the correct path.

Due to of their deep pipeline and high issue rates, future processors will speculatively execute many instructions prior to branch resolution. While this speculation enables instruction issue to continue during branch computation, it produces some unavoidable side effects. One effect is an increase in the number of instruction and data references. These extra, “wrong” references could cause cache pollution and significantly increase memory bus traffic. On the other hand, they could act as prefetching references, bringing data or instruction lines into the cache for later, correctly predicted path execution. Another effect stemming from speculation is that only valid writes should modify the cache since they must be easily undone during misprediction recovery. These changes in the execution model warrant a reexamination of current memory system models. In this paper we examine the effects of deep speculation on cache behavior.

The ideal method for examining the cache performance of a speculative processor is to generate memory reference traces with a full execution simulator and use them as input to a cache simulator. Unfortunately, execution simulators are time consuming both to build and run. As an alternative, we have designed a code instrumentation tool called spex which runs on Intel architecture (ix86) Unix systems and can be used to generate memory traces which are a close approximation to that of a speculative processor. Long traces can be obtained relatively quickly without the use of an execution simulator. Spex adds branch prediction, speculative execution, and memory trace production to an existing binary. It creates a new binary which can be run on a current generation, non-speculative processor. The number of cycles required to resolve each conditional branch is approximated with a fixed value, n, which is an input to the tool. In reality, branch resolution times in pipelined and superscalar machines will vary depending upon the type of conditional branch, the data dependencies between instructions currently in the pipe-

line, the number, type, and data dependencies of instructions waiting to be issued, and processor exceptions. Thus, our results should be interpreted as bounds for machines that speculate no more than n instructions past a conditional branch.

Since the spex model and a true speculative processor have different viewpoints of speculative paths, it is important to clarify some terms used in this paper. A speculative path is the execution path taken after a conditional branch prediction. The spex model can immediately classify the speculative path as a correct path (correctly predicted direction) or a wrong path (mispredicted direction) by comparing the predicted direction with the actual result of the branch. Thus it can tag references generated on these paths as correct path references or wrong path references. It must be remembered, however, that a real speculative processor cannot foresee the accuracy of the prediction until the branch is resolved some distance down the speculative path. Therefore, an immediate cache action such as line allocation cannot be based upon the correctness of the reference.

We used spex and several different cache simulators to model cache performance when speculatively executing SPEC92 applications. It was not our intent to specify an optimal cache configuration for a certain processor based only upon application generated references. Many studies have shown that this leads to meaningless conclusions [5]. Instead we examined the differences in cache performance between the speculative and non-speculative execution models. We believe that similar results would be found using a more complete workload containing OS effects.

The main result of the study is that deep speculation causes a significant increase in the number of data references, yet data misses increase by usually less than 15%. In fact, by calculating the traffic ratio we found that cache efficiency actually increases as speculation increase. Further investigation revealed that wrong path misses exhibit prefetch, pollution, and LRU reordering effects. In most observed cases the prefetch effect is dominant and wrong path misses cause a decline in the number of correct path misses. In addition, it is observed that mispredicted path instruction references also exhibit this prefetching effect. Finally, we discuss write allocation policies and the number of buffers required to hold speculative writes.

## 2 Description of Spex

Spex is a binary instrumentation tool which allows the user to gather speculative traces on a non-speculative processor. It was built for an Intel architecture (i386, i486, Pentium) computer running Unix SysVR4 and currently will instrument statically-linked code produced by the Intel/AT&T and USL CCS C compilers. Spex and other trace generation tools add additional instructions to a program to record runtime information and memory references while preserving the original logical operation of the program [10][12]. In addition, spex incorporates branch prediction and wrong path recovery code to allow the execution of mispredicted paths. In use, spex is similar to that of IDtrace for the Intel architecture processors and pixie for the MIPS processor family [6][9]. For input, spex requires a binary file and two numbers. One number specifies the choice of branch prediction algorithm and the other number is the branch resolution depth. A depth argument of zero corresponds to no speculation while a positive argument represents the number of instructions to be executed down a mispredicted path before recovery. Since all mispredictions take the same number of instructions to resolve, the case of a later conditional branch being resolved before an earlier one cannot occur.

To date, seven prediction algorithms have been implemented and they can be divided into four classes. Three are static algorithms: always taken, prediction based on branch opcode, and prediction based on opcode and direction. Two are dynamic predictors using a variable sized history table. The table is indexed by the address of the conditional branch. It can be direct-mapped, up to eight-way set associative, or a table entry can be shared by multiple addresses. One dynamic algorithm has a bit per table entry corresponding to the previous direction of the branch. The other has a two or three bit saturating counter per table entry to maintain a weighted history for each branch [8]. Another group of prediction algorithms use two-level adaptive training schemes with various size history tables and registers [13]. The final type is a profile algorithm where the direction is determined by looking for the prediction entry in a file. If an entry for a particular branch doesn't exist in the file, one of the above algorithms is used to predict the branch.

The memory trace generated by spex is a list of 5-byte entries (1 byte tag, 4 byte address), one for each instruction or data memory reference. Different tags are used to differentiate between correct path and wrong path references. Most generated traces are extremely long and therefore are piped directly to a memory simulator. As an example of use, spex can instrument a program called bench by typing:

```
spex -d 10 -b 5 bench
```

where `bench` is a statically-linked executable. The new executable, `bench.spex`, will execute 10 instructions down a wrong path and use branch prediction algorithm number 5 (dynamic counter with a default counter size of 2 bits). Running `spex.bench` creates two more files: `bench.spst`, a statistic file containing runtime information and `bench.sptr`, the trace file. These files can then be

used by postprocessing tools or the trace file can be piped into a memory system simulator.

### 2.1 Instrumentation Details

The executable file structure and memory reference instrumentation is based on that of a previous i486 instrumentation tool, IDtrace. In short, the text section is disassembled, binary code is inserted before every instruction, all control instructions are relocated to account for the text expansion. Then, the new text, the original, unmodified data sections, some working tables, and trace buffers are combined to create the new executable. Complex ix86 instructions, indirect addressing, and variable instruction lengths are some of the issues which make binary instrumentation difficult and sometimes impossible. In-depth discussion of these issues and some restrictions on applicable programs can be found in [6]. The speculative instrumentation adds significantly to the size and runtime of the new binary. Code expansion is roughly 25 times. Runtime is increased by factors of 25-45 for zero depth (no speculation) and 35-65 for depth 10 speculative execution. In comparison, IDtrace produces a binary about 12 times larger and increases the runtime by about a factor of 12 for a full execution trace. Spex adds code prior to each conditional branch to call the prediction algorithm and possibly begin wrong path execution. Code must also be added before each instruction to possibly exit from wrong path execution. Finally, guard instructions must be added prior to any instruction that can cause an exception. This prevents an exception from occurring and halting execution while processing incorrect data down a wrong path. For instance, suppose during execution of an instrumented program a pointer has yet to be initialized and it has the value zero. The original code might be of the form

if (*pointer initialized*)
then *reference pointer*
else *initialize pointer*

and the conditional branch corresponding to the *if* statement could be mispredicted. Then an attempt would be made to reference memory location zero and a segmentation fault would halt execution. Boundary checking code is added before all reads and writes to prevent this error. Unfortunately, there are many more exceptions such as divide by zero and floating point errors which could be triggered by incorrect data during wrong path execution. Since it is too cumbersome to add code to check for all exceptions only segmentation fault protection code is added throughout the instrumented code. Deep path speculation on some programs causes floating point errors and code is added on an individual basis to handle these cases.[1]

The instrumented code action is straightforward. Assume a correct path is being executed. At each branch the prediction algorithm is called and the result is compared with the known actual direction of the branch. If they are the same execution continues down the correct path. If the branch is mispredicted, the register state and correct next instruction location are saved, a counter is set to the wrong path depth argument, and execution begins down the wrong path. The following actions are taken for each wrong path instruction:

- The counter is decremented and when it is zero wrong path execution is terminated.
- If the instruction performs a read or write the address is first checked against segment boundaries to prevent segment violations caused by incorrect data.
- Memory reference entries are output with special wrong path tags.
- If the instruction performs a write, the address and original value are stored in a write restore buffer.

When wrong path execution terminates the writes are undone using the data stored in the write restore buffer, the register state is restored, the prediction algorithm is updated, and execution restarts at the saved correct branch path address. Sometimes wrong path execution can terminate without going down the full wrong path depth. This can happen in the following cases:

- Exit call - Cannot exit during wrong path execution.
- System call - This is a call to the OS and cannot be traced.
- Data segment fault - An invalid read or write outside the data segment would cause a fault if allowed to proceed.
- Indirect jump - An attempt to index a jump table with an invalid index will usually cause execution to continue outside the text section and so wrong path execution is always halted.
- Indirect call - Indirect calls in instrumented code are handled by a runtime lookup matching the original target address with the new target address in the instrumented code, see [6]. If the original address is computed from incorrect data the lookup will fail and wrong path execution is halted.
- Execution fault - This could be caused by a floating point or divide by zero exception. Code cannot be added before every instruction to test for all possible exceptions so complete execution terminates.

The above termination conditions occur very infrequently. In the benchmarks described below, wrong paths

1. The C signal library functions could have been used to handle all exceptions efficiently but its use would have interfered with breakpoint exceptions and made debugging binary code extremely difficult.

executed to the full depth of 25 instructions over 90% of the time, see Table 1. Furthermore, it is unlikely that a real speculative processor would continue issuing instructions down a path which produced a segmentation fault. Thus, premature terminations do not compromise the accuracy of the trace.

| Termination Type | Occurrences | Percentage |
|---|---|---|
| System Call | 347K | < 1% |
| Data Segment Fault | 14.1M | 8% |
| Indirect Jump | 947 | 0% |
| Indirect Call | 13K | 0% |

**TABLE 1.** Termination statistics -The number and percentage of mispredicted paths which were terminated before executing down 25 instructions. Statistics are for the combination of all benchmarks. The total number of mispredicted paths was 174.8 million.

## 3 Results

To study the effect of speculative execution on cache performance, traces were generated by spex instrumented code and fed into two cache simulators: a modified version of Tynero and a multicache simulator designed to monitor prefetching and pollution effects [7]. Both simulators distinguish between correct path references and misses and wrong path references and misses. The test suite consisted of the SPEC92 C benchmarks, see Table 2. All experiments were run on an Intel 50 MHz i486 machine running USL Unix SysVR4.2.

| Program | Description |
|---|---|
| cc1 | Major forked process of GNU C Compiler |
| compress | Unix compression utility |
| ear | Inner ear model, floating point |
| eqntott | Boolean equation to truth table translator |
| espresso | Logic minimization tool |
| sc | Spreadsheet Program |
| xlisp | XLISP interpreter solving 8 queens problem |

**TABLE 2.** The SPEC92 benchmarks used in this study.

Our processor/memory model is a multi-issue, pipelined processor with out-of-order execution requiring deep speculation to maintain a high instruction issue rate. All mispredicted paths are assumed to execute down a constant, predefined depth before branch resolution. The processor has a first level set-associative, non-blocking cache with an LRU replacement policy. Cache line size is fixed at 32 bytes for all experiments. The cache completes all outstanding memory requests. If a wrong path reference causes a read miss,the cache line is updated even if the branch is resolved before the request to memory is completed. Speculative path writes are held until the branch is resolved so that the cache always contains valid data. Finally, the cache write policy is copy-back with write allocation for correct writes.

Two branch prediction algorithms are used in this study. Algorithm 1 is a two-level adaptive scheme with a 512 entry, 4-way associative register table and a 4096 entry pattern table containing 2-bit saturating counters [13]. This algorithm is expensive in terms of hardware but it achieves excellent accuracy for the benchmarks in our study, see Table 3. It is the algorithm used unless otherwise specified. Algorithm 2 uses a simpler history table of 1024, 2-bit saturating counters. No tags are recorded so multiple addresses which map to the same table entry will share the counter.

| Program | Alg. 1 | Alg. 2 | Program | Alg. 1 | Alg. 2 |
|---|---|---|---|---|---|
| cc1 | 88 | 85 | espresso | 93 | 86 |
| compress | 89 | 87 | sc | 96 | 92 |
| ear | 96 | 94 | xlisp | 95 | 85 |
| eqntott | 96 | 82 | | | |

**TABLE 3.** Branch prediction accuracies.

### 3.1 Total Data Traffic

The main result of our study is that the total data memory traffic is not significantly increased by deep speculation in most benchmarks. Figure 1 shows the percent increase in total (correct and wrong path) misses of speculative execution over that of non-speculative execution.

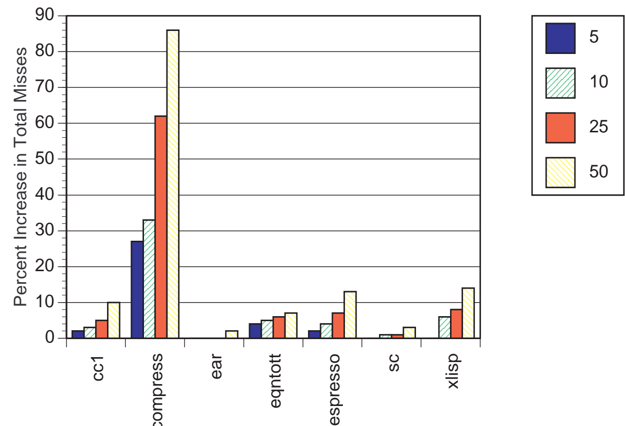


**FIGURE 1.** Speculative data miss percentage increase over that of non-speculative execution. The cache is 32K with 4-way set associativity.

Figure 2 shows that there is a substantial increase in the number of data references due to speculation (up to 75%

for 50 deep speculation) yet for most benchmarks deep speculation increases the total number of data cache misses by only around 15% over non-speculative data cache misses. The traffic ratio, defined as

$$\text{traffic ratio} = \frac{(\text{cache misses} + \text{copy-backs}) * (\text{cache line size})}{(\text{memory references without a cache}) * (\text{word size})}$$

is a measure of the efficiency of the cache. Figure 3 shows that the cache usually becomes more efficient, traffic ratio decreases, as speculation increases. The exception to this is compress. A 0.71 traffic ratio without speculation from Figure 3 reveals that the cache is not well suited for this application. Repeating the experiment using a smaller line size resulted in a lower miss ratio. Furthermore, 8% of the non-speculative references missed the cache when speculating 25 instructions deep. We suspect that compress has poor locality down several mispredicted paths which repeatedly get executed. The large line size magnifies the traffic problem. More work is required to find which instructions in compress produce these misses.

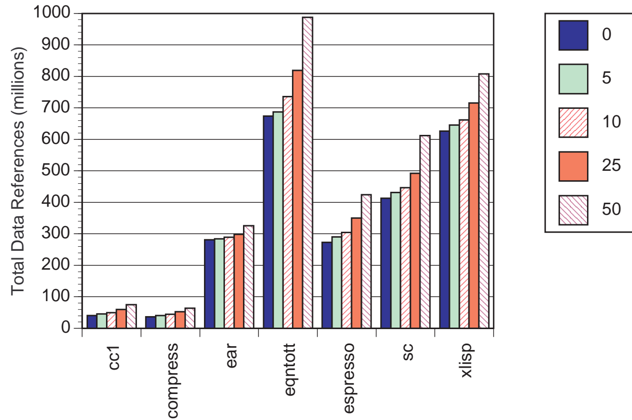


**FIGURE 2.** Total data references. The zero depth speculation represents no speculation.

The next two figures display how the bandwidth increase changes with increasing cache size and increasing associativity. As would be expected, configuration changes which reduce pollution generally reduce the additional bandwidth required for speculative execution. The inconsistent data for large associativities is probably due to the small number of misses.

## 3.2 Wrong Path Miss Effects

There are several ways that the additional wrong path references can affect the cache. First, they can prefetch data which will later be used during correct path execution. A miss during correct path execution can then be avoided if this data is accessed before being displaced in the cache.

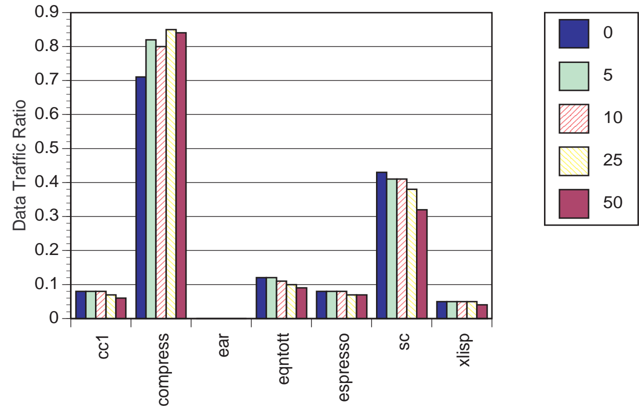


**FIGURE 3.** Traffic ratio for different benchmarks and speculation depths. The cache size is 32K and is 4-way set associative.

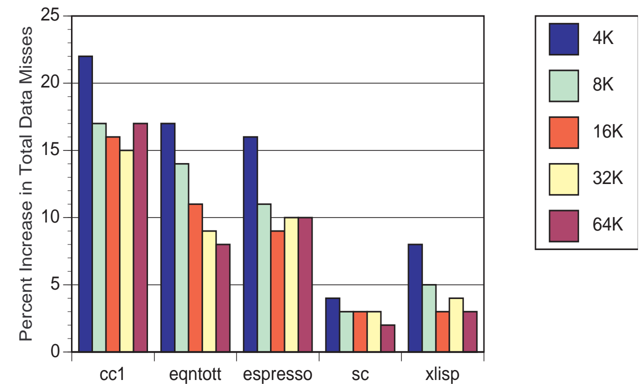


**FIGURE 4.** Bandwidth increase over non-speculative execution for different cache sizes. Caches are direct mapped and the speculation depth is 25.

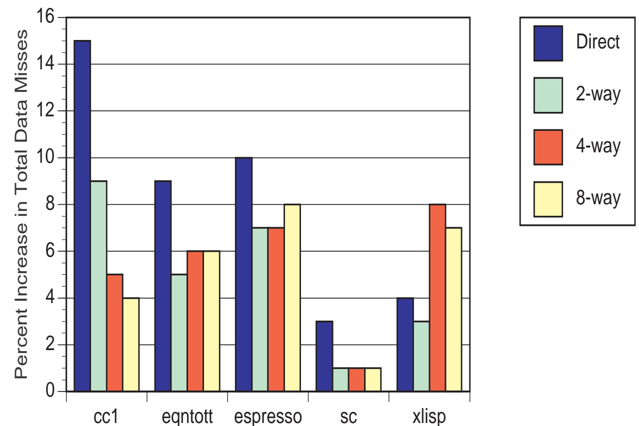


**FIGURE 5.** Bandwidth increase over non-speculative execution for different cache associativities. Caches are 32K and the speculation depth is 25.

These wrong path misses are called prefetch misses and they reduce the number of correct path misses. On the other hand, pollution misses increase the number of correct path misses. They are caused by wrong path read misses allocating lines which displace lines needed for later correct path

execution. To discuss these effects it is helpful to define a ratio $P_n$, where

$$P_n = \frac{\text{\# of correct path misses for a given cache with speculation depth } n}{\text{\# of non-speculative misses for a given cache}}$$

Figure 6 gives $P_n$ for speculation down 25 instructions using the higher accuracy prediction algorithm. It shows that speculation reduces the number of correct path misses so prefetching must dominate the pollution caused by wrong path reads. Deeper speculation increases the number of wrong path references, thereby increasing the prefetch misses and further reducing $P_n$. Another way to increase the number of wrong path references is to use a less accurate prediction algorithm to execute more wrong paths. Figure 7 shows a more dramatic reduction in $P_n$ due to using the less accurate Algorithm 2. Notice in compress, $P_n$ increases with depth signaling that pollution, rather than prefetch, is the dominant effect.

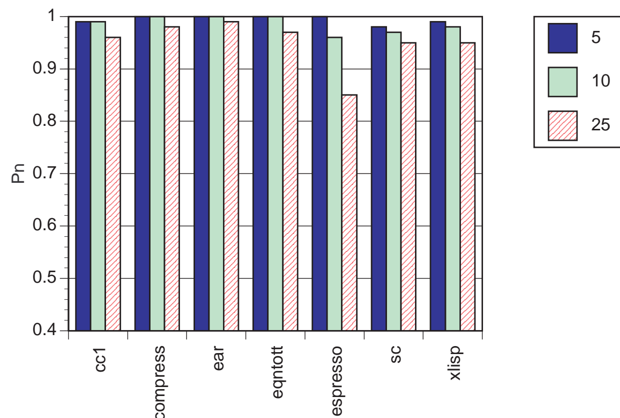


**FIGURE 6.** $P_n$ for different speculation depths using branch prediction algorithm 1. The cache is 16K with 4-way set associativity.

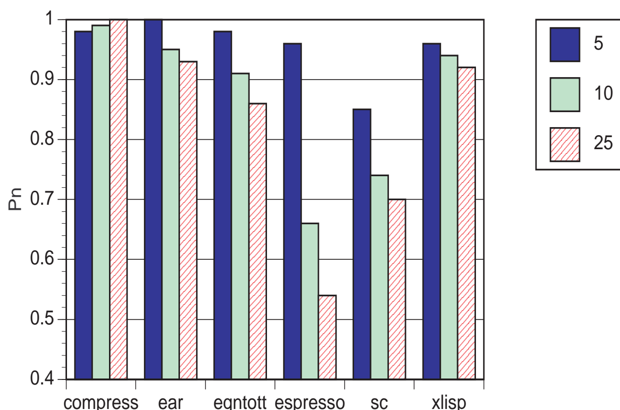


**FIGURE 7.** $P_n$ for different speculation depths using branch prediction algorithm 2. The cache is 16K with 4-way set associativity.

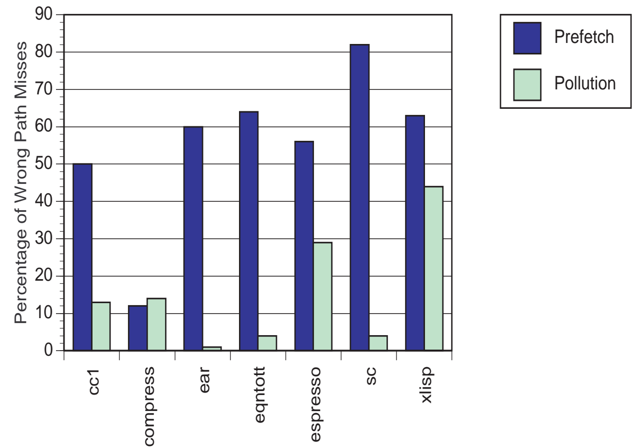


**FIGURE 8.** Breakdown of prefetch and pollution effects. The cache size is 16K with 4-way set associativity.

The cache simulator was modified to directly count the number of prefetch and pollution misses caused by wrong path references and it was found that they alone did not completely account for the change in $P_n$. It was observed that wrong path cache hits can also reduce (or increase) correct path misses. They do so by reordering the lines in a LRU set associative cache. For example, suppose that a wrong path reference hits a least recently used line and thus promotes it to most recently used. Then suppose a cache read miss occurs in the same set. A different line will be displaced than if execution were non-speculative. If the line is needed during correct path execution this action will have avoided a miss. If the displaced line rather than the promoted one is needed an additional miss is incurred. Reordering has only a secondary effect on cache misses compared to that of prefetch and pollution. Figure 8 shows the percentage of wrong path misses which were prefetch or pollution misses. Notice that, with the exception of compress, over 50% of misses performed a prefetch.

## 3.3 Wrong Path Writes

During speculative execution our processor model must delay all write references occurring down a speculative path until the branch is resolved. This requires write buffers between the processor and cache to hold the address and value of these writes until branch resolution. If the branch was predicted correctly the writes are released to the memory system. Otherwise, the suspended writes are squashed. Figure 9 shows that most instructions are issued speculatively so most writes will be temporarily suspended. This leads to several questions. How many buffers are necessary to hold most wrong path writes and what write allocation policy for the cache is appropriate?

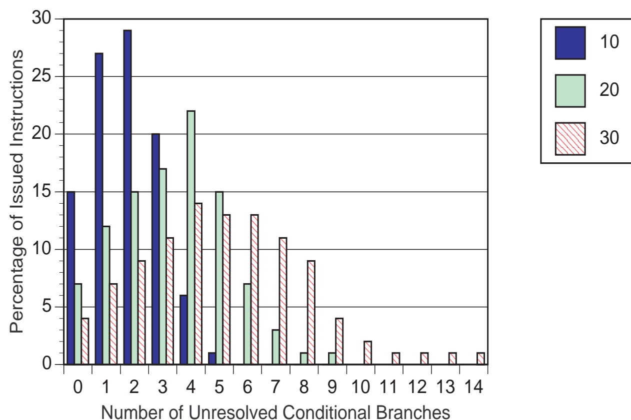

**FIGURE 9.** Percentage of instructions issued per number of outstanding conditional jumps.

### 3.4 Speculative Write Buffers

The instrumented benchmarks were run and the number of writes in speculative paths were counted to estimate the number of needed buffers. Figure 10 shows the results. To fully execute 90% of the speculative paths, 4, 6, and 10 write buffers are required for depths of 10, 20, and 30 respectively. Paths with more write references than available buffers can still partially complete before instruction issue must be halted.

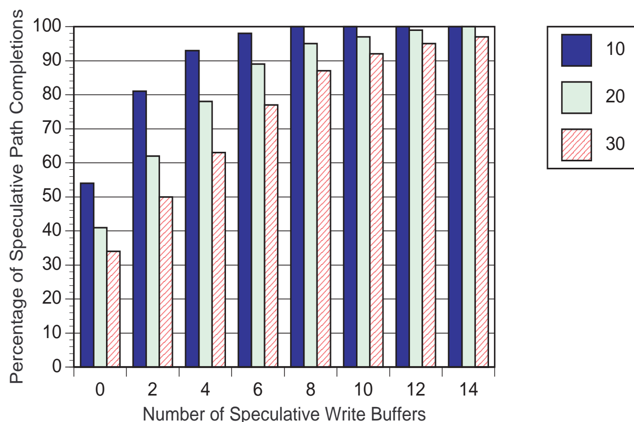

**FIGURE 10.** Percentage of speculative paths fully executed for various numbers of speculative write buffers.

### 3.5 Wrong Path Write Allocate

Most caches which use a copy-back write policy also implement write allocate, i.e., a write miss causes the line to be allocated in the cache. Write allocation on a speculative processor can be handled in several ways. One way is to wait until the branch is resolved and allocate lines for the correct writes in the cache write buffers. Another way would be to allocate cache lines for writes as they are being suspended during speculative execution. This would have the desirable effect of prefetching the lines so that some lines could be in the cache at resolution time. But writes produced by mispredicted path will also cause write allocations and this has an unclear overall effect on cache performance. Allocations produced by mispredicted paths which are never taken during correct execution will increase memory traffic and cache pollution. However, wrong path write allocation might also prefetch lines for later correct path execution. Figure 11 compares the effects of allocating cache lines for all speculative writes or just for committed writes after branch resolution. The data shows allocating all writes increases the total memory traffic by a negligible amount. Furthermore, allocating wrong path writes performs a small amount of prefetching for later correct path memory references. Therefore, it is beneficial to allocate cache lines on all speculative write references prior to branch resolution.

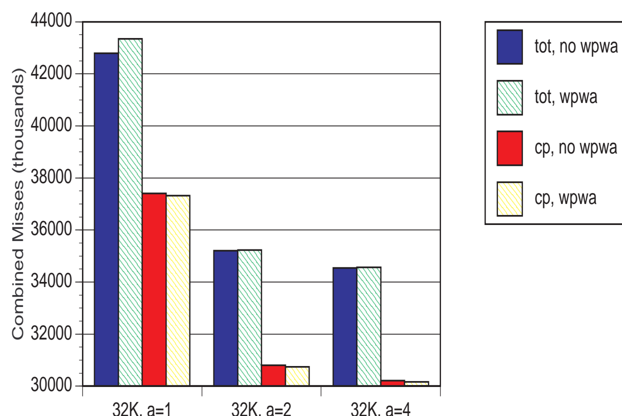

**FIGURE 11.** Effect on memory traffic when wrong path writes are allocated in the cache.

### 3.6 Instruction Prefetching

This study has focused on data cache behavior because the SPEC benchmarks do a poor job of exercising even small instruction caches [2]. Pollution and prefetch effects cannot be observed when the full working set fits into the cache. However, we believe that instruction references produced during mispredicted paths could perform prefetches for later instruction references. Table 4 shows that over 50% of the lines which were allocated in the instruction cache during mispredicted paths were then accessed during correct path execution later in the program. Thus, half of all wrong path instruction misses could prefetch useful instruction cache lines. Since few conflict misses occur in the non-speculative runs of these applications, it is probable that the pollution effect of the unused half would be minimal. However, the additional number of lines allocated due to executing mispredicted paths is small and will get smaller with better prediction. An area of further study is the potential benefit of aggressively prefetching lines down wrong paths. Since accurate prediction algorithms limit the

number of wrong path executions this would entail allocating instruction cache lines from both the taken and not taken execution paths.

| Program | Reuse % | Program | Reuse % |
|---|---|---|---|
| cc1 | 60 | espresso | 68 |
| compress | 52 | sc | 56 |
| ear | 66 | xlisp | 60 |
| eqntott | 60 | | |

**TABLE 4.** Wrong path instruction reuse - The percentage of instruction cache lines allocated during wrong path execution which were later referenced during correct path execution.

## 4 Conclusion

We have developed an instrumentation tool to generate approximate speculative execution trace streams without the need for an execution simulator. With this tool we were able to compare the cache performances of speculative and non-speculative execution models on a suite of application benchmarks.

From this study, insight was gained into how the increased number of memory references due to speculation affect cache performance. While wrong path misses do contribute to cache pollution, their dominant effect is to prefetch cache lines for later correct path execution. During wrong path execution both data and instruction misses allocate useful information over 50% of the time. Although the actual amount of prefetching is small due to the infrequency of wrong path execution, wrong path prefetching could lead to a viable method of prefetching instruction and data cache lines in non-numeric code. The major result of the work is that deep speculation does not significantly increase data memory traffic in most of the tested applications. Thus, it is not likely to pose a serious problem for the cache designs of future speculative microprocessors exploiting a high degree of instruction-level parallelism.

## Acknowledgments

We would like to thank Intel Corp. for its support and especially Konrad Lai for his guidance and suggestions throughout this work.

## References

[1] D. Alpert, and D. Avnon, “Architecture of the Pentium Microprocessor,” IEEE Micro, June 1993, pp. 11-21.

[2] J. Gee, M. Hill, D. Pnevmatikatos, A. Smith, “Cache Performance of the SPEC Benchmark Suite,” Technical Report UCB/CSD 91/648, University of California Computer Science Division, Berkeley, CA.

[3] R. Groves, and R Oehler, RISC System/6000 Processor Architecture, IBM RISC System/6000 Technology, SA23-2619, IBM Corporation, 1991, pp. 16-24.

[4] E. McLellan, “The Alpha AXP Architecture and 21064 Processor,” IEEE Micro, June 1993, pp. 26-47.

[5] D. Nagle, R. Uhlig, T. Stanley, T. Mudge, S. Sechrest and R. Brown, “Design Tradeoffs for Software-Managed TLBs,” *Proc. of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 27-38.

[6] J. Pierce, “IDtrace: A Trace Generation Tool for the ix86 Instruction Set,” Technical report, Dept. of Electrical Engineering and Computer Science, University of Michigan, Dec. 1993.

[7] J. Quinlan, and K. Lai, “Tynero: A Multiple Cache Simulator,” Technical Report, Intel Corp., Hillsboro, OR, May 1991.

[8] J.E. Smith, “A Study of Branch Prediction Strategies,” *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981, pp. 135-148.

[9] M. Smith, “Tracing with Pixie,” Technical Report, Center for Integrated Systems, Stanford University.

[10] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. Shen, “Instruction Level Profiling and Evaluation of the IBM RS/6000,” *Proc. of 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, 1991, pp. 180-189.

[11] Sun Microsystems Laboratories, Inc., “Introduction to SpixTools,” Technical Report, Mountain View, CA, April 1992.

[12] D. Wall, “Systems for Late Code Modification,” Digital Western Research Laboratory, Research Report, June 1991.

[13] T-Y Yeh, and Y. Patt, “Two-Level Adaptive Training Branch Prediction,” *The 24th ACM/IEEE International Symposium and Workshop on Microarchitecture*, Nov. 1991, pp. 51-61.