

Performance Analysis of Wrong-Path Data Cache Accesses

R. Iris Bahar [†] and Gianluca Albera ^{§†}

[†] Brown University
Division of Engineering
Providence, RI 02912

[§] Politecnico di Torino
Dip. di Automatica e Informatica
Torino, ITALY 10129

Abstract

The performance of today's high-end microprocessors continues to grow. This increase in performance is due in part to the use of speculative, out-of-order execution, coupled with highly accurate branch prediction. However, even with branch prediction accuracies over 90%, many instructions are executed unnecessarily from the wrong path. This wrong-path execution results in cache pollution and unnecessary bus traffic.

We use an extended version of the SimpleScalar simulation tool to model an out-of-order, 4-way issue processor. First, we use various branch confidence mechanisms to determine if a data access is likely on a mis-predicted path or not. Next, we sort the data cache accesses using these confidence mechanisms and compare the "inherent re-usability" of data that is marked either "low confidence" (i.e. likely on the wrong path) to those marked "high confidence" (i.e. likely on the correct path). Finally, with the assistance of small buffers associated with the first level data cache, we try to increase this re-usability factor by preventing low confidence data accesses from being written into the data cache. We discuss the effectiveness of this technique as well as the behavior and reuse of speculative data.

1 Introduction

Today's high-end microprocessor performance increase is due to many factors; among them is the use of speculative, out-of-order execution with highly accurate branch prediction. Branch prediction has increased instruction-level parallelism (ILP) by allowing programs to speculatively execute beyond control boundaries, while out-of-order execution has increased ILP by allowing more flexibility in instruction issue and execution. The combination of these techniques has increased processor performance in part by hiding the memory latency penalty in the case of a cache miss. Although speculative execution is essential for increasing the instructions per cycle (IPC), a large amount of unnecessary work results from "wrong-path" operations in the pipeline [5]. Wrong path operations occur when the branch predictor mispredicts a branch. In particular, a measurable number of unnecessary memory accesses result from wrong-path execution.

Unnecessary accesses to the memory subsystem can

create problems in terms of performance. First, the access is utilizing resources (namely the memory ports and buses) which could have been used by other (useful) accesses. Second, assuming a multi-level memory hierarchy, wrong-path memory accesses may displace useful data from the caches. In other words, the cache may become *polluted* with potentially useless data.

In this paper, we analyze the effects of wrong-path execution on cache performance. Specifically, we will report on the percentage of accesses to the data cache that are speculative down a wrong path and determine how much of a factor "wrong" or "right" path execution may make on the re-usability of data. This work differs from previous work on wrong-path effects on caches in two ways. First, our work is based on wrong path effects on the data instead of the instruction cache, as was done in previous work by Pierce and Mudge [12]. In their work, the authors showed that wrong path instruction prefetching may have beneficial effect in terms of processor performance. While this may be true for instructions, our experiments have found that it is less true for the data. The second difference is how we deal with these findings.

We make the claim that wrong path data is more likely to pollute the cache with non-reusable data than correct path data. Based on this claim, we present a method to alleviate some of the cache pollution problems created by wrong-path operations. We introduce a 16-entry, fully associative structure used to store wrong path data accesses. Since in a real CPU it is not possible to determine if instructions are on the wrong path until the branch is resolved, we introduce a *confidence* mechanism to estimate if memory accesses are likely on the mispredicted path [1, 4, 5]. When the machine is estimated to be executing wrong-path instructions, the memory subsystem places all fills from the second level cache into this structure (called the *confidence buffer*) instead of the first level main cache. During correct path operations, fills are normally stored in the main cache. This *confidence buffer* is accessed in parallel along with the first level cache to determine a hit or miss.

The rest of this paper is organized as follows. In Section 2 we describe the hardware architecture we used, as well as the experimental environment and the mechanisms used to determine if an access is likely down a wrong path. In Section 3 we present detailed experi-

mental results and in Section 4 we conclude and indicate possible future work.

2 Experimental Setup

This section describes the hardware mechanism and experimental environment we use in this paper. We refer to the confidence estimation described in [1, 4, 5] to determine when the pipeline is in a *low-confidence* state (i.e. likely mispredicted) and, according to this estimate, we make use of *confidence buffers*. Finally, we present the extended version of the SimpleScalar tool set [2, 3] and the configuration we use for our simulation.

2.1 Confidence Estimation

Current branch prediction technology can achieve a prediction accuracy of between 65% and over 90% [7, 8]. This results in a large overhead of unnecessary instructions and unnecessary memory accesses. Branch confidence estimation attempts to classify each branch prediction as having “high confidence” or “low confidence” [6, 4]. A “high confidence” classification means that the branch was likely predicted correctly, and a “low confidence” classification means that the branch was likely mispredicted. Note that the confidence classification of high and low confidence is independent of the branch prediction of branch taken or branch not taken.

2.1.1 “Both Strong” Confidence Estimator

A detailed analysis of confidence estimation metrics and methods is presented in [4]. We used the results from this work to determine the best confidence classification method for using buffers in a cache architecture. Since we needed to capture a large amount of the mispredicted branches, we decided to use the “Both Strong” method in conjunction with the combined branch predictor proposed by McFarling [7]. Variations of the combined branch predictor are used in the Pentium Pro [9] and the new Alpha 21264 chip [10]. The McFarling combined branch predictor uses two different branch predictors and a predictor-predictor to determine which method to use for the current branch. Each branch predictor has a table of counters to indicate the prediction based on the behavior of the last N branches. The state machine for the branch prediction counters is given in Figure 1. The counters are 2 bits each, and the value of the counter for each state is given in parentheses. The “Both Strong” confidence estimator determines the confidence of the branch based on the state of the counters. If both counters are in a strong state, and in the same strong state, the estimator labels this branch as a high confidence branch. Table 1 shows the truth table for estimating confidence based on the value of the branch prediction counters.

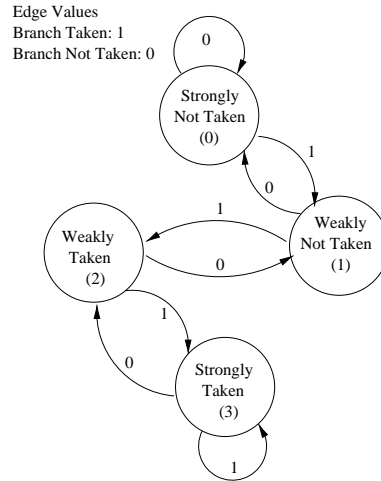


Figure 1: State machine for 2-bit branch prediction counters. The McFarling combined branch predictor uses two branch predictors, each using a set of 2-bit counters.

2.1.2 “Either Strong” Confidence Estimator

A variation of the “Both Strong” estimator, called “Either Strong”, is also presented in Table 1; in this case we mark a branch as high confidence, if either one of the counters is in the strong state, low confidence otherwise. Now more branches will be classified as high confidence, and there will be a higher probability that what we mark low confidence is really on the wrong path. On the other hand, we may be mislabeling a larger number of wrong path instructions as “high confidence”.

Table 1: State table for “Both Strong” and “Either Strong” confidence estimators based on McFarling counter values.

Counter1	Counter2	<i>Both Strong</i>
Weak	Weak	LowConf
Weak	Strong	LowConf
Strong	Weak	LowConf
Strong	Strong	If same state, HighConf Else LowConf

Counter1	Counter2	<i>Either Strong</i>
Weak	Weak	LowConf
Weak	Strong	HighConf
Strong	Weak	HighConf
Strong	Strong	HighConf

2.1.3 Secondary Filter Mechanism

Unfortunately, the level of accuracy is small for the “Both Strong” predictor, i.e., only about 21% of

branches labeled low confidence are truly mispredicted. To compensate for this lack of accuracy, we added the secondary filter mechanism described in [5]. The secondary filter mechanism estimates the state of the pipeline based on the number of unresolved, low confidence branches in the pipeline at one time. If there are M low confidence branches in the pipeline at one time, and each low confidence branch has a 20% probability of being mispredicted, then the probability that at least one of the M branches has been mispredicted becomes $1 - (1 - 0.20)^M$. Hence, when $M = 4$, the probability of a mispredicted branch somewhere in the pipeline becomes 59%. The confidence threshold T represents the minimum number of low confidence branches necessary before a cache access is labeled *low confidence*. If $M \geq T$, then we mark the access as *low confidence*, otherwise *high confidence*. A unique counter is used to keep track of the number of low confidence branches.

We saw that the “Either strong” method does not need a secondary filter, since it marks low confidence only those branches that have a high probability of being mispredicted. Note that for the data cache, store operations are never marked as low confidence, since they are committed (i.e. written in the cache) only when we are sure we are executing on the correct path.

2.2 Simulator Environment

All the simulations were completed using an extension of the *SimpleScalar* tool suite [2, 3]. SimpleScalar is an execution-driven simulator that uses binaries compiled to a MIPS-like target. SimpleScalar can accurately model a high-performance, dynamically-scheduled, multi-issue processor. We use an extended version of the simulator that more accurately models all the memory hierarchy, implementing non-blocking caches and complete bus bandwidth and contention modeling [3]. Other modifications were added to handle precise modeling of cache fills.

Tables 2, 3, and 4 show the configuration of the processor modeled. Note that first level caches are on-chip, while the unified second level cache is off-chip. In addition we have a *confidence buffer* associated with the first level data cache; these buffers are 16-entry fully associative, with LRU replacement. Note that we chose an 8K first level cache configuration in order to obtain reasonable hit/miss rate from our benchmarks [11]; the size of the *confidence buffer* has been chosen as a tradeoff between improved performance and timing/area/power issues.

Our simulations were executed on SPECint95 benchmarks; they were compiled using a re-targeted version of the GNU *gcc* compiler, with full optimization. Since we are executing a full model on a very detailed simulator, the benchmarks take several hours to complete. Therefore, due to time constraints, we feed the simulator with a small set of inputs; however, we execute all programs entirely.

Table 2: Machine configuration parameters.

Parameter	Configuration
L1 Icache	8KB direct; 32B line; 1 cycle lat.
L1 Dcache	8KB direct; 32B line; 1 cycle lat.
L2 Unified Cache	256KB 4-way; 64B line; 12 cycles
Memory	64 bit-wide; 20 cycles on page hit, 40 cycles on page miss
Branch Pred.	2k gshare + 2k bimodal + 2k meta
BTB	1024 entry 4-way set assoc.
Return Addr. Stack	32 entry queue
ITLB	32 entry fully assoc.
DTLB	64 entry fully assoc.

Table 3: Processor resources.

Parameter	Units
Fetch/Issue/Commit Width	4
Integer ALU	3
Integer Mult/Div	1
FP ALU	2
FP Mult/Div/Sqrt	1
DL1 Read Ports	2
DL1 Write Ports	1
Instruction Window Entries	64
Load/Store Queue Entries	16
Fetch Queue	16
Minimum Misprediction Latency	6

Table 4: Latency and occupancy of each resource.

Resource	Latency	Occupancy
Integer ALU	1	1
Integer Mult	3	1
Integer Div	20	19
FP ALU	2	1
FP Mult	4	1
FP Div	12	12
FP Sqrt	24	24
Memory Ports	1	1

3 Experimental Results

3.1 Base Case

In this section we will describe the base case we used. As stated before, it uses 8K direct mapped first level caches (i.e. DL1 for data and IL1 for instruction), with an off-chip unified 256K 4-way second level cache (UL2). Table 5 shows the execution time measured in cycles and, for each memory level (DL1, IL1, UL2, and Main Memory (MM)), the number of accesses and misses; all these values are given in thousands. For clarity we also give the miss rate (*m-rate*). Notice that on average accesses to the data cache miss 7.7% of the time (and at most 13.1% for *go*).

Table 5: **Base case results.** Accesses, misses and miss-rate are given for each memory level. Values are in thousands except for *m-rate*.

Name	Cycles	DL1			IL1			UL2			MM
		acc.	misses	m-rate	acc.	misses	m-rate	acc.	misses	m-rate	acc.
compress	70 538	27 734	2 338	8.4 %	130 560	10	0.01 %	3 132	284	9.08 %	492
go	826 111	169 851	22 351	13.1 %	954 702	36 175	3.79 %	60 056	280	0.47 %	290
vortex	250 942	93 470	7 557	8.9 %	214 249	9 070	4.23 %	15 336	355	2.32 %	522
gcc	392 296	103 490	7 529	7.8 %	361 405	17 616	4.87 %	22 966	376	1.64 %	427
li	134 701	74 453	5 574	7.9 %	288 335	1 117	0.39 %	5 856	4	0.08 %	5
jpeg	139 376	73 354	3 778	5.5 %	307 159	818	0.27 %	3 876	96	2.48 %	144
m88ksim	659 451	130 319	3 289	2.2 %	479 488	32 764	6.83 %	34 953	12	0.04 %	19
perl	372 034	90 363	7 971	8.1 %	305 208	17 486	5.73 %	24 531	142	0.73 %	174

Table 6: **Inherent Re-usability in data cache.**

Test	Oracle					Both Strong $T = 3$				
	Inh. reuse		hit / fill behavior			Inh. reuse		hit / fill behavior		
	Lconf	Hconf	LC acc.	LC fill	Hit LC fill	Lconf	Hconf	LC acc.	LC fill	Hit LC fill
compress	38.0 %	60.4 %	2.83 %	13.89 %	2.18 %	40.0 %	61.2 %	7.85 %	18.92 %	6.36 %
go	53.7 %	66.7 %	8.54 %	18.41 %	7.54 %	57.1 %	68.4 %	23.39 %	37.27 %	19.28 %
vortex	39.0 %	52.9 %	0.07 %	0.21 %	0.12 %	44.0 %	53.2 %	2.07 %	4.99 %	1.42 %
gcc	50.4 %	74.1 %	3.77 %	10.57 %	2.36 %	61.3 %	74.9 %	10.48 %	22.53 %	8.09 %
li	56.9 %	88.1 %	2.81 %	3.52 %	1.34 %	85.9 %	86.9 %	4.27 %	10.54 %	3.34 %
jpeg	66.0 %	81.9 %	0.83 %	4.24 %	2.13 %	70.1 %	82.7 %	5.43 %	13.80 %	6.45 %
m88ksim	90.4 %	65.8 %	0.60 %	2.29 %	0.90 %	37.1 %	68.9 %	1.67 %	7.15 %	1.46 %
perl	46.6 %	74.3 %	2.32 %	7.03 %	1.21 %	49.7 %	76.3 %	7.01 %	16.41 %	4.58 %

3.2 Inherent Re-Usability

Our work started with the assumption that line fills coming from wrong path misses have a smaller “inherent re-usability” (i.e. have a lower probability of being further accessed), thus they are more likely to contribute to cache pollution. That is, they are introduced to the cache without being used and replace other blocks that are more likely to be useful. Table 6 shows this re-usability factor for the data cache using two different sorting schemes. The out-of-order model in the SimpleScalar tool set has the capability of determining which instructions are from the wrong path at the time of decode. We call this the **Oracle**, since it implements a perfect confidence predictor (i.e. marks all true wrong path accesses as *low confidence*). We also present results using the *confidence* scheme, using **Both Strong** and a threshold $T = 3$, since it gives, overall, best results. We do not present results using the “Either Strong” mechanism for space constraints, but results were analogous to the “Both Strong” case. We also limit our analysis to data cache behavior for the same reason.

Results in Table 6 show two different characteristics of data reuse. The first one, called *inherent reuse* (columns 2–3 and 7–8), keeps track of the percentage of low and high confidence fills that were accessed again at least once. If blocks are not re-accessed, we waste cache entries, since we replace potentially useful blocks with other ones that will not be accessed again. We ana-

lyzed this behavior for block fills marked *low confidence* and *high confidence*. In the **Oracle** section we show that line fills due to wrong path accesses have a lower inherent re-usability than the ones due to correct path (e.g., for *compress* 38% vs. 60.4% respectively). The average re-usability for low confidence is 55%, while high confidence is 68% (see columns 2 and 3 of Table 6). This trend also extends to the **Both Strong** section, where we try to estimate the wrong/correct path using the *confidence* scheme previously introduced. The re-usability of low confidence fills generally increases (compared to the Oracle). This can be expected, since we are only doing an estimate; thus some correct path memory accesses may end up being treated as if they were on the wrong path and vice-versa.

In the previous paragraph we showed the behavior of blocks that are accessed more than once, but we didn’t differentiate whether the block was accessed again only once or several times. This “amount of reuse” is the second characteristic we analyze, and we call it *hit / fill behavior* (columns 4–6 and 9–11). Column *LC acc.* is the percentage of memory accesses marked as *low confidence*; *LC fill* is the percentage of low confidence accesses that missed, thus generating a cache fill; and *Hit LC fill* is the percentage of all hits that are coming from the low confidence fills we just mentioned. For example if we consider *compress* in the **Oracle** case, we see that only 2.83% of all memory accesses are from the

Table 7: **Data cache results - Oracle.** Percent improvement over the base case for when using the *confidence buffer* with an oracle.

Test	Percentage decrease compared to the base case				confidence buffer & re-usability information					
	ΔC_{yc}	$\Delta DLI_{acc.}$	ΔDLI_{miss}	$\Delta UL2_{acc.}$	Inh. reuse		hit / fill behavior			%spared
					Lconf	Hconf	LC acc.	LC fill	Hit LC fill	
compress	1.04 %	-0.07 %	7.38 %	6.62 %	36.03 %	60.59 %	2.76 %	10.91 %	14.51 %	4.30 %
go	2.85 %	1.11 %	24.50 %	10.77 %	55.17 %	68.72 %	7.56 %	19.20 %	10.11 %	13.29 %
vortex	0.68 %	0.11 %	6.77 %	3.76 %	72.44 %	51.74 %	0.08 %	0.30 %	25.11 %	2.17 %
gcc	1.04 %	0.16 %	15.59 %	5.15 %	55.38 %	74.27 %	3.51 %	10.25 %	9.60 %	6.24 %
li	1.67 %	0.08 %	6.95 %	6.64 %	68.38 %	88.76 %	2.76 %	5.63 %	6.24 %	2.79 %
jpeg	2.62 %	0.47 %	13.94 %	13.95 %	48.45 %	81.99 %	0.44 %	1.95 %	11.13 %	5.37 %
m88ksim	0.23 %	0.09 %	22.22 %	2.29 %	62.93 %	61.76 %	0.52 %	2.24 %	17.37 %	8.04 %
perl	0.27 %	-0.05 %	8.58 %	3.07 %	55.08 %	74.55 %	2.30 %	7.77 %	5.06 %	3.82 %

wrong path, but they are responsible for 13.89% of the overall number of misses. This means that wrong path accesses are more likely to miss. Furthermore, even if these blocks account for 13.89% of the fills, they account only for 2.18% of the overall hits, showing that they present a smaller reuse than correct path fills. This behavior for the Oracle is also observed for the **Both Strong** case show in columns 7–11.

3.3 Confidence Buffers

As shown in Section 3.2, we found that low confidence misses fill the cache with less useful data, thus polluting the cache. We tried to avoid this negative effect by introducing a *confidence buffer*; we put low confidence fills in this buffer and thereby avoid replacing more useful blocks in the main cache.

Table 7 shows the effects of the *confidence buffer* using the **Oracle** case. Columns 2–5 of the Table show a comparison with the base case in terms of performance and cache behavior. Numbers given are percentage decreases compared to the base case; thus positive numbers will mean a reduction, while negative numbers an increase. Note that values for DL1 accesses and misses now refer to the combined effect of the primary data cache and *confidence buffer* (e.g. we have a miss if and only if the access misses in both the DL1 and the *confidence buffer*.)

The reduction in data cache misses (ΔDLI_{miss}) is significant (13.2% on average and up to 24.5% for *go*). However this does not directly translate into performance gain; for example *m88ksim* has a 22.22% reduction in the number of misses, but only a 0.23% performance improvement. In this particular case we can partially explain this behavior considering the very small miss rate in the base case: 2.2% for *m88ksim* (see Table 5). However, we also see programs such as *go*, that have a 13.1% miss rate in the data cache for the base case and a reduction of 24.5% in miss rate, but a gain of only 2.85% in performance. We suspect that the misses we avoided were predominantly on the wrong path; if this was true, we reduced the latency of instructions that

eventually were discarded (as soon as the mispredicted branches were resolved). This only partially explains the disparity between the miss reduction and performance improvement; we will further discuss why miss rate does not translate to performance gain later in this paper.

Columns 6–10 of Table 7 have the same meaning of those previously explained in Table 6. However, now that we have a buffer to store low confidence data, reading them requires more attention. Some programs, such as *go*, show an increase in re-usability for high confidence blocks that are now contained in the primary data cache (*Inh. reuse - HConf* 68.72% vs. 66.7% shown in Table 6). However, this increase is quite small and does not seem to fully explain, for example, the 24.50% reduction in misses we have in *go*. To better understand what happens consider first the column labeled *Hit LC fill* for *go*. We see that the number of hits coming from the low confidence data is 10.11% compared to 7.54% for the base case, showing a better use of this data thanks to positioning it in the confidence buffer. Now consider the column labeled *%spared*; this column gives the percentage of hits that would have missed if we didn't have the confidence buffer. That is, we keep track of the number of entries in the main cache that have been “spared” from replacement due to the use of the *confidence buffer* and are later accessed again. Looking at this column, we see that 13.29% of hits would not have happened in the main cache if we didn't move some fills to the buffer. In summary, we see that we improve our cache behavior in a double manner. First, we make better use of the main cache by removing some low confidence fills and thus sparing replacements and misses. Second, we also increase the number of hits from low confidence fills, since they are now stored in a separate buffer. Note that even if these blocks were filled by wrong path instructions, this does not mean they won't be accessed by correct path instructions in the future.

Another point is that we cannot simply compare the base case values shown in Table 6 with the case using a buffer. First, we are moving part of the fills to a 16-entry fully associative buffer, which has a distinct behavior from a 256-entry direct mapped cache. In addi-

Table 8: **Data cache results - Both Strong, with Threshold = 3.** Percent improvement over the base case when using a *confidence buffer* and the “Both Strong” sorting mechanism.

Test	Percentage decrease compared to the base case				confidence buffer & re-usability information					
	ΔC_{yc}	$\Delta DLI_{acc.}$	ΔDLI_{miss}	$\Delta UL2_{acc.}$	Inh. reuse		hit / fill behavior			%spared
					Lconf	Hconf	LC acc.	LC fill	Hit LC fill	
compress	1.90 %	-0.04 %	11.57 %	11.54 %	26.61 %	61.09 %	7.53 %	7.82 %	63.00 %	7.83 %
go	3.41 %	1.38 %	29.20 %	13.05 %	55.20 %	74.54 %	21.19 %	34.23 %	12.58 %	19.37 %
vortex	2.39 %	-0.07 %	25.26 %	11.14 %	65.32 %	55.59 %	1.94 %	3.66 %	21.69 %	8.07 %
gcc	1.51 %	0.21 %	21.79 %	7.27 %	60.79 %	76.12 %	9.88 %	21.06 %	9.72 %	11.03 %
li	2.44 %	0.12 %	11.14 %	10.50 %	90.92 %	87.83 %	4.11 %	6.34 %	14.05 %	4.45 %
jpeg	3.33 %	0.52 %	20.55 %	19.55 %	69.27 %	82.20 %	4.82 %	12.03 %	10.93 %	6.89 %
m88ksim	0.29 %	0.14 %	32.71 %	3.17 %	33.99 %	69.00 %	1.62 %	4.03 %	8.57 %	15.17 %
perl	0.66 %	0.09 %	12.56 %	4.47 %	46.64 %	76.38 %	6.73 %	14.14 %	7.89 %	6.59 %

tion, the combined effects of these two caches may lead to a completely different behavior. For example *vortex* has a negligible number of accesses marked low confidence (0.08%), that contribute to 0.3% of cache fills; nevertheless these fills account for 25.11% of the overall number of hits. Since the buffer is fully associative and uses LRU replacement, it offers extra associativity for storing some extremely useful (i.e. frequently accessed) data.

This behavior is also evident in Table 8, which shows results using the “Both Strong” confidence predictor. Here *compress* has 63% of its hits coming from the *confidence buffer*, implying that there are a few blocks of data that happened to be placed in the buffer and frequently accessed. In this case the gain we obtain in terms of miss reduction is no longer related to the speculation mechanism, but instead due to the extra associativity offered by the buffer. We can say that in these cases, the confidence buffer acts in a similar manner as a victim cache [13].

Another fact that supports this victim cache behavior is that in the case of “Both Strong” we mark a larger number of instructions as low confidence, compared to the Oracle (see column *LC acc.*). By doing so, we introduce in the buffer some fills that we estimate being low confidence, but that actually are on the correct path. Despite this lack of precision, we see that miss reduction (and performance) is better than the Oracle. Since we generally have a very small number of memory accesses that are from the wrong path (an average of 2.5%), in the Oracle case we don’t fully exploit the buffer capacity to store data, so by adding some extra fills, we can increase the “associativity” of the main cache and obtain a further improvement.

The gain in performance, like in the Oracle case, is somewhat disappointing (up to 3.41% and 2% on average), but comes with a good improvement in miss rate (up to 32% and 20.6% on average). Other than the reasons previously explained, we tried to further explain this fact by running a set of benchmark with an infinite data cache (i.e. after the cold start effect, it always hits). Programs like *go* and *vortex* showed a 17% improvement in performance, while others like *perl* and

gcc had only a 12% increase and *m88ksim* was as small as 2.5%. This may indicate that SPECint95 benchmarks have a small sensitivity to data cache latency, thanks to the out-of-order execution that, matched to particular code behavior, partially hides the miss latency. It is unclear whether other programs (especially commercial workloads) would present the same behavior.

4 Conclusions and Future Work

In this paper we analyzed the re-usability behavior of low confidence (i.e likely on wrong path) data, versus high confidence, (i.e. likely on correct path) data. We also presented a simple mechanism to assign confidence to memory accesses and the use of *confidence buffers* to reduce cache pollution and consequently improve performance. For all the examples we ran, performance improved on average 2% (and up to 3.41%). While the performance gain is somewhat disappointing, it may indicate two things. First, although low confidence data has a lower rate of re-usability, the difference is not enough to make a significant impact in performance if used as a sorting mechanism. Secondly, note that for the SPECint95 benchmarks at most only 7.56% of accesses are actually from wrong path instructions and the average is much lower (2.5%). Even with a perfect predictor, we will be sorting out only a very small number of accesses and several of these may end up being accessed again anyway. It would be interesting to see how confidence sorting may improve performance for other applications which may exhibit pathological wrong path behavior.

Since we observed that, in some circumstances, the *confidence buffers* also present a *victim cache* behavior [13], we are currently investigating the possibility of combining these two techniques. First results show that it is possible to improve performance further by combining both these schemes.

Acknowledgments

We wish to thank Bobbie Manne for her invaluable help and support throughout this work. We also wish to thank Doug Burger for providing us with the framework for the SimpleScalar memory model.

References

- [1] R. I. Bahar, G. Albera and S. Manne, "Using Confidence to Reduce Energy Consumption in High-Performance Microprocessors," to appear in *International Symposium on Low Power Electronics and Design (ISLEPD)*, 1998
- [2] D. Burger, and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report TR#1342, University of Wisconsin, June 1997.
- [3] D. Burger, and T. M. Austin, "SimpleScalar Tutorial," presented at *30th International Symposium on Microarchitecture*, Research Triangle Park, NC, December, 1997.
- [4] D. Grunwald, A. Klauser, B. Manne, A. Pleszkun, "Confidence Estimation for Speculative Control," to appear in *ISCA-25: ACM/IEEE International Symposium on Computer Architecture*, June 1998.
- [5] S. Manne, D. Grunwald, A. Klauser, "Pipeline Gating: Speculation Control for Energy Reduction," to appear in *ISCA-25: ACM/IEEE International Symposium on Computer Architecture*, June 1998.
- [6] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions," *MICRO-96: ACM/IEEE International Symposium on Microarchitecture*, pp. 142–152, December 1996.
- [7] S. McFarling, "Combining Branch Predictors," Digital WRL Research Report TN-36, June 1993.
- [8] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," *ISCA-20: ACM/IEEE International Symposium on Computer Architecture*, pp. 257–266, May 1993.
- [9] Intel Corporation, Pentium-Pro data book, 1996.
- [10] L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report*, October, 1996. <http://www.digital.com/semiconductor/microrep/digital2.htm>
- [11] Jeffrey Gee, Mark Hill, Dinoisions Pnevmatikatos, Alan J. Smith, "Cache Performance of the SPEC Benchmark Suite," *IEEE Micro*, Vol. 13, Number 4, pp. 17-27 (August 1993)
- [12] J. Pierce, and T. Mudge, "Wrong-Patch Instruction Prefetching," *IEEE Micro*, December, 1996.
- [13] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *ISCA-17: ACM/IEEE International Symposium on Computer Architecture*, pp. 364–373, May 1990.