# 15-740: Computer Architecture
# Final Report
# Speculation and Cache Problems in STAMPede

Patrick Riley <pfr+@cs.cmu.edu>
Joseph Vanderwaart <joev+@cs.cmu.edu>
Kevin Watkins <kw+@cs.cmu.edu>
http://www.cs.cmu.edu/~kw/15-740

Fall 2000
Submitted: December 4, 2000

## 1 Introduction

Steffan et. al. [3] have described a technique based on *thread-level speculation* [1, 4] whereby non-parallel code can be compiled and executed on a parallel architecture with little or no effort on the part of the programmer. Techniques for doing so have long existed for applications exhibiting uniform, regular data access patterns, so that the compiler can identify regions of code which might be executed in parallel and statically analyze all data dependences between them. However, outside the scientific computing and digital signal processing domains such applications are rare.

The solution Steffan et. al. describe is to augment the parallel machine with extra hardware which dynamically detects data dependence violations as they occur. This frees the compiler to optimistically identify regions of code which might be parallelized without exhaustive static knowledge of their data access patterns. These regions correspond to dynamic instruction sequences, *epochs*, which are forked onto individual processors at runtime. An epoch is executed speculatively while the possibility exists that a data dependence on a logically earlier epoch will be executed in the wrong order. If such a *dependence violation* is discovered, the effects of the speculative epoch will be squelched and the epoch will be reexecuted.

This has been proposed as a general framework applicable both to single-chip multiprocessors and to more general distributed memory multiprocessing configurations. However, a question remains as to how to efficiently maintain information about the speculatively executed effects of speculative epochs, and how to detect violations based on that information. The elegant solution of Steffan et. al. is to exploit the invalidation-based cache coherence mechanism of the parallel architecture to store speculation information as well. Speculative modifications to memory are only allowed to propogate down to a certain level of the memory hierarchy, so that they can be squelched easily in the event of a violation. RAW and WAW data dependence violations can be detected via an extension to the ordinary invalidation mechanism. Briefly, an invalidation message is accompanied by the *epoch ID* of the epoch generating it. Epochs receiving the message must suffer a violation if they have speculatively loaded or modified the same cache line and are logically later than the epoch generating the invalidation. Further details may be found in Steffan et. al. [3].

## 2 Coping with Replacement

One issue that arises in using a modified cache coherence protocol to track speculatively loaded or modified data is what to do when a cache line containing speculation information is about to be evicted. Unless special precautions are taken, an epoch which is about to evict a speculatively loaded or modified cache line must incur a *replacement violation* and be reexecuted. Since this is not a true dependence violation, however, it would also be possible to *stall* the offending epoch until all earlier epochs have committed (in other words, until the offending epoch receives the *homefree token*). At that point all the epoch's speculative cache lines become non-speculative, and so the cache line that was to be evicted can be replaced as usual. Both of these approaches are described by Steffan et. al. [3].

Steffan and Mowry [4] describe a further optimization: instead of stalling or reexecuting an epoch that is about to replace a cache line containing speculation information, we could move such lines into a special *victim cache*. With a moderately associative regular cache and in the absence of unfortunate aliasing behavior, the number of such replacements that might be expected to occur in a single epoch is low. Steffan and Mowry present experimental data suggesting that in many cases a two-way set-associative regular cache coupled with a four entry victim cache is sufficient to avoid nearly all replacement violations. An issue they do not discuss is that the replacement policy for the regular cache can be chosen to prefer evicting non-speculative lines before evicting speculative ones. Note that the LRU policy is not always sufficient in the presence of compiler hints which allow a speculative epoch to perform non-speculative loads or stores.

In order to confirm the results presented by Steffan and Mowry as to the number of replacement violations that might be expected given regular caches of various associativities and victim caches of various sizes, as well as to assess the performance impact that unmitigated replacement violations might be expected to have, we extended their simulation framework to support a victim cache. We did not consider the effect of suspending replacement violations or modifying the replacement policy of the set-associative cache.

## 3 Experimental Setup

The utility of a victim cache was evaluated through simulation. For each processor, a 4-way issue, out-of-order, superscalar processor similar to the MIPS R10000[5] was simulated. We simulated a

| | |
|---|---:|
| Cache Line Size | 32B |
| Instruction Cache | 32KB, 4 way set associative |
| Data Cache | 32KB, 2 banks |
| Unified Secondary Cache | 2MB, 4 banks, 4 way set associative |
| Miss Handlers | 8 for data, 2 for instructions |
| SMURF: crossbar interconnect | |
| Minimum Miss Latency to Secondary Cache | 12 cycles |
| Minimum Miss Latency to Local Memory | 75 cycles |
| Main Memory Bandwidth | 1 access per 20 cycles |

Table 1: Simulation memory parameters

| Application | Input Data Set | Speculative Region | Unrolling Factor | Avg. Intrs. Per Epoch | Parallel Coverage |
|---|---|---|---|---|---|
| `compress95` | test [test.in] | compress.c:480, while loop | 1 | 196.7 | 24.6% |
| | | compress.c:706, while loop | 1 | 240.4 | 22.7% |
| `ijpeg` | test [specmun.ppm] quality 10 smoothing_factor 10 | jccolor.c:138, for loop | 32 | 1467.9 | 8.2% |
| | | jcdctmgr.c:214, for loop | 1 | 80.8 | 2.2% |
| | | jidctint.c:171, for loop | 1 | 84.0 | 5.0% |
| | | jidctint.c:276, for loop | 1 | 100.3 | 6.7% |

Table 2: Speculatively parallelized regions

tightly coupled single chip multiprocessor in all of our experiments. The full memory system was faithfully simulated with separate data and instruction caches for each processor at the first level connected by a crossbar to a unified, shared second level cache. The memory system values (which were fixed throughout the experiments) are shown in Table 1.

We modified the simulator described in [3, 2] in order to support a victim cache. We varied the number of processors (2, 4, 6, or 8), the associativity of the first level data cache, and the size of the victim cache. We used two benchmarks from the Spec95 suite in evaluation: `compress95` and `ijpeg`. The regions of these benchmarks that were annotated for TLDS are shown in Table 2 (taken from [3]).

## 4 Results

We were concerned with two types of violations: DINV, referring to data dependence violations, and REPL, referring to replacement violations. As mentioned earlier, previously published papers on thread-level data speculation [2, 3] have stated that a significant number of epochs must be restarted due to REPL violations: that is, eviction of speculatively read or modified lines from the cache. We were unable to confirm these results, as when we ran simulations of both the `ijpeg` and the `compress95` benchmarks we did not observe any such violations even with only a 2-way set associative cache. Even with a direct-mapped data cache, the number of REPL violations was small compared to the total number of epochs simulated, and so the potential for improving performance through the use of a victim cache was smaller than expected. Nevertheless, we did observe that the addition of even a small victim cache was helpful both in eliminating REPL violations and in speeding up the overall execution time.

The results of our simulations are shown in Figures 1 through 3 and Table 3. Since neither `compress95` nor `ijpeg` produced any REPL violations when run on a simulated machine with a two-way set associative data cache all the graphs refer to a direct-mapped cache.

Figure 1 shows the effect of the size of the victim cache on the number of REPL violations that occurred. Note that a one-line victim cache was sufficient to avoid all of the REPL violations for `compress95`, while for `ijpeg` two lines were required. In fact, in a direct-mapped data cache with a very large victim cache (1K entries), `compress95` never required more than one victim cache line per processor per epoch, and `ijpeg` always used either zero or two victim cache lines at a time. Figure 2 shows a view of the effect of a one-line victim cache on all violations, not just REPL violations. The violations produced by the benchmarks are divided into *primary* and

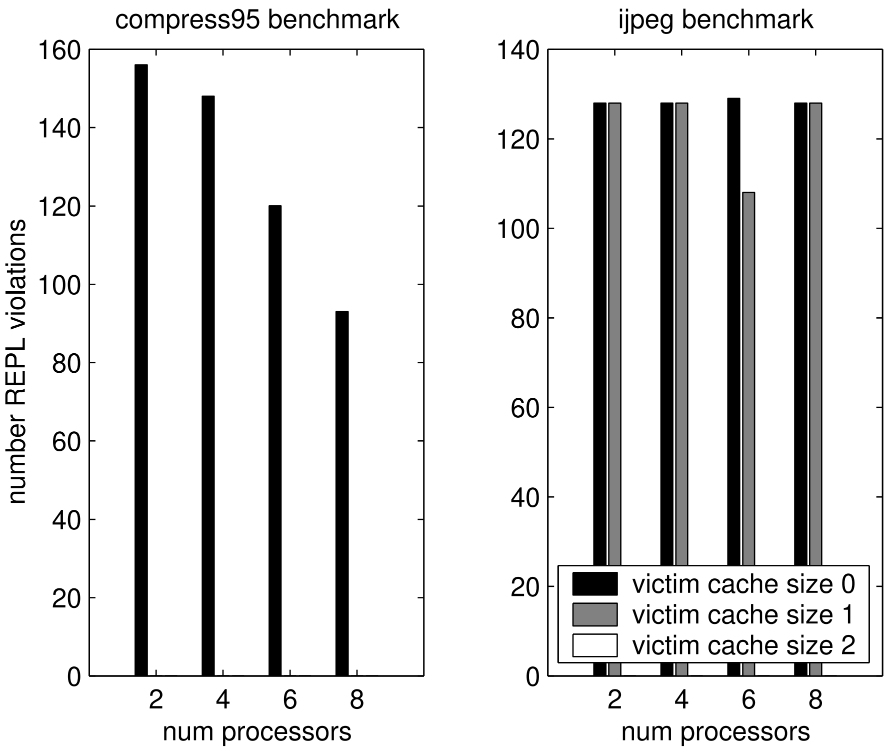


Figure 1: Replacement violations with different victim cache sizes.

*secondary* violations. Primary violations are those that occur when neither the affected epoch nor any logically earlier one has suffered any violations since beginning to execute; secondary violations are either violations that occur after a primary violation in the same epoch or the so-called CHAIN violations that kill all logically later epochs when one epoch violates. As would be expected based on the observed victim usage, the most significant effect a one-line victim cache had on `ijpeg` was a decrease in *secondary* violations: for instance, any REPL that had been the primary violation for a given epoch in the baseline experiment was no longer a violation at all, and the first secondary violation in that epoch became the primary one. Interestingly, even though none of the epochs simulated for `ijpeg` used only one victim line when two or more were available, it still was able to avoid some primary REPL violations using a one-line victim cache.

It is interesting to note that for both benchmarks, reducing the number of REPL violations by adding a victim cache also reduced the number of DINV violations. The reasons for this were not immediately obvious, but we will discuss a possible explanation in the next section. Another difficult observation to account for is that the set of lines moved to the victim cache seems to be the same in every epoch that uses it at all. That is, it appears there is only one cache line in `compress95` that is ever placed in the victim cache, and there are only two such lines in `ijpeg`. It is impossible to know the reason for this behavior without careful analysis of the source code for the benchmarks.

Finally, the effect of the victim cache on total execution time is depicted in Figure 3 and Table 3. Note that in every case the addition of a victim cache sufficiently large to prevent REPL violations

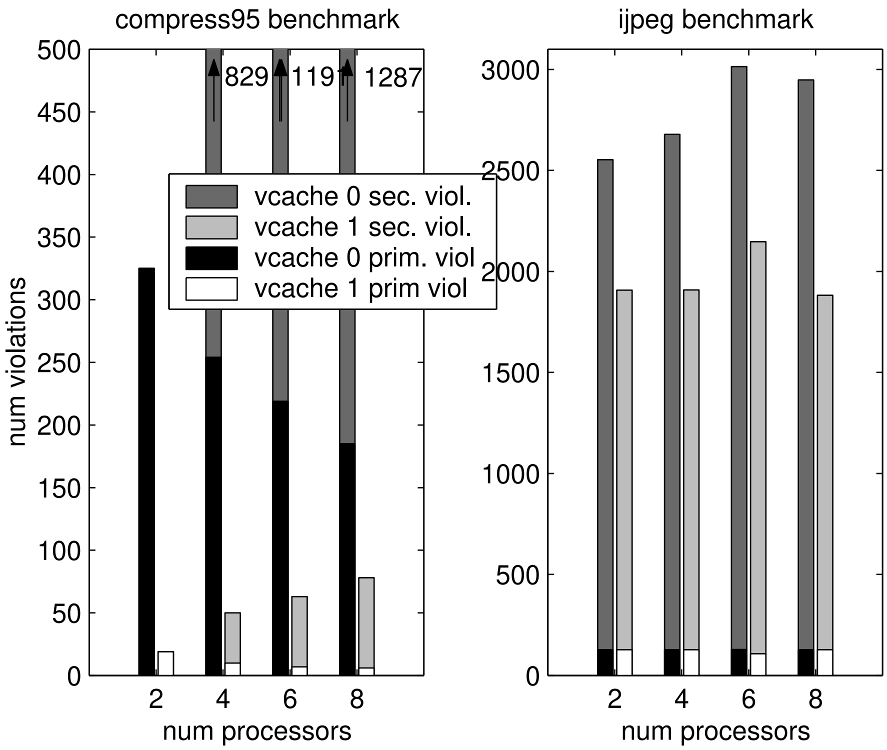


Figure 2: Primary vs. secondary violations in 0- and 1-line victim caches.

led to a small but always beneficial change in execution time (*i.e.*, the speedup was greater than one in every case). Speedups for the `compress95` benchmark averaged 1.0039, while those for `ijpeg` ranged from 1.0003 on 6 processors to 1.0007 on 4.

One curiosity in the results is that for the `compress95` benchmark, adding a single victim cache entry (for any number of processors) reduces the number of DINV violations as well as the number of REPL violations. At first it is not clear why a REPL violation should have any effect on whether a DINV violation occurs. A possible explanation is as follows: A violated epoch continues to run until the end of its instructions even after a violation occurs. This means that a single epoch in a single run can cause multiple violations, such as a REPL and a DINV violation. The epoch may have to be rerun several times in order to avoid a REPL violation, in particular, the epoch must become non-speculative for it to run correctly. This can increase the apparent number of DINV violations. Further, REPL violations can change the "window" of epochs which are running at a particular time, changing whether data dependencies between epochs will cause violations.

## 5 Conclusion

We have certainly confirmed our hypothesis that a small victim cache is sufficient to eliminate replacement violations in the kind of thread-level speculation system we have been discussing. However, at least two factors indicate that the victim cache may nonetheless be overkill. First, the actual number of replacement violations even for direct-mapped or not very associative cache

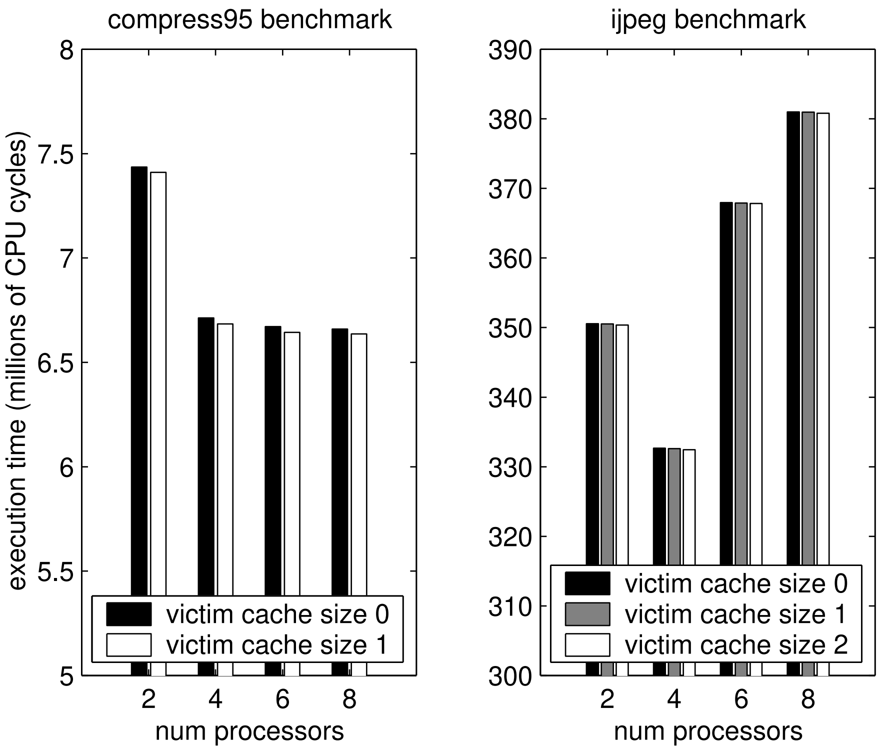


Figure 3: Total execution time for varying numbers of processors and victim cache sizes.

setups was much lower than we suspected. Second, the actual runtime impact of replacement violations was never significant. This can be explained by the fact that allowing moot epochs to run to completion may act to prefetch data, warming up the cache for their eventual reexecution.

Questions which remain for future investigation are whether these results scale to the more loosely-coupled multi-chip distributed memory architectures considered by Steffan et. al. [3] and what the effect (if any) of modifying the replacement policy of the set associative cache to treat speculative lines preferentially would be.

## References

[1] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of ISCA 22*, pages 414–425, June 1995.

[2] J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Extending cache coherence to support thread-level data speculation on a single chip and beyond. Technical Report CMU-CS-98-171, Carnegie Mellon University, Pittsburgh, PA, December 1998.

[3] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of ISCA 27*, June 2000.

| Processors | Time, no VC | Time, 1 VC line | Speedup | Time, 2 VC lines | Speedup |
|---|---|---|---|---|---|
| `compress95` | | | | | |
| 2 | 7.436 | 7.410 | 1.0035 | | |
| 4 | 6.713 | 6.684 | 1.0043 | | |
| 6 | 6.671 | 6.644 | 1.0041 | | |
| 8 | 6.660 | 6.636 | 1.0036 | | |
| Avg. | | | 1.0039 | | |
| `ijpeg` | | | | | |
| 2 | 350.572 | 350.526 | 1.0001 | 350.362 | 1.0006 |
| 4 | 332.669 | 332.624 | 1.0001 | 332.441 | 1.0007 |
| 6 | 367.964 | 367.898 | 1.0002 | 367.840 | 1.0003 |
| 8 | 380.995 | 380.961 | 1.0001 | 380.790 | 1.0005 |
| Avg. | | | 1.0001 | | 1.0005 |

Table 3: Execution times (in millions of processor cycles) of benchmarks with and without victim caches.

[4] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[5] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.