Shared Cache Extension to Thread-Level Speculation

CS15-740 Project Report

Andrew Faulring, Yan Karklin

December 8, 2000

1 Abstract

We extended the Thread-Level Speculation (TLS) enabled Cello simulator to allow multiple speculative processors to share a single cache. Previously Cello required that all caches with speculative state be private — each cache can have only one processor above it in the memory hierarchy. We ran three benchmarks to compare the performance among different shared and private cache configurations. Our performance metric was the number of processor cycles required to complete the benchmark. Shared caches performed better than private caches on all the benchmarks tested. In fact, a 16 processor configuration ran the ijpeg benchmark slower with private caches than with a shared cache that was sixteen times smaller than the total size of the private caches. We posit that the speculative threads are exhibiting cooperative rather than competitive memory access patterns.

2 Introduction

With the rapidly decreasing transistor sizes, computer architects designing uniprocessors face dimishing returns in terms of performance due to electrical and physical limits. Parallel architectures can utilize the increasing transistor capacity, hopefully improving performance past the uniprocessor barriers. Soon, one might imagine that multiprocessors will replace the ubiquitous uniprocessor. Modern operating systems already schedule multiple processes or theads to run concurrently on different processors. In order to reap maximum benefit from parallelized program execution, two issues must be addressed: concurrency must be found and utilized by assigning different tasks to different processors; at the same time, the resulting communication and data transfer costs must be kept minimal.

Thread-Level Speculation (TLS) has been used to extract parallel threads from traditional sequential programs (often irregular and non-numeric in nature). A tightly coupled multiprocessor executes separate iterations of the same loop, while a modified cache allows the speculative iterations to proceed without affecting the system's permanent state until all earlier speculative iterations have successfully completed. The benefits of thread-level speculation on various aspects of performance and the implications for cache coherence protocols have been recently reported by Steffan *et al.* [1].

We extend this research by investigating the performance issues related to allowing separate processors to share a single cache. Recent work suggests that shared cache implementations outperform multiprocessor systems with private caches for some memory request and locality patterns [2].

If data is truly shared across processors, then programs with good spatial and temporal locality should benefit from the shared cache implementation. When a single processor registers a read miss and brings a line into the shared cache, any other processors subsequently reading that cache line should have the fetching latency hidden. On the other hand, concurrent threads using the same cache could contend to bring different lines into the cache, causing data to ping-pong in and out of the cache, which increases latency and hurts performance.

In order to investigate the impact of a shared cache on performance, we extended the TLS-enabled Cello simulator to allow multiple processors to share a single cache. In the next section, we describe the modifications necessary for this extension, as well as their possible impacts on performance. In the results section, we compare the performance among shared cache configurations and private cache configurations, evaluating several configurations of the multiprocessor shared cache setup.

3 Implementation

Extending the Cello simulator to support shared cache speculation required three changes. First, the cache line state was extended to support one pair of speculatively loaded and speculatively modified bits per processor. Second, the cache transition function was updated to support accesses by multiple processors, some speculative. Third, the memory system was extended by adding per processor states, such as ownership required buffers.

With instruction level parallelism (ILP), the processor hides the speculative state using register renaming and other internal buffers. The processor can manage the speculative state because it only executes a couple dozen instructions ahead. However, with thread level parallelism (TLP) the granularity of parallelism increases significantly, and the processor may need to speculatively execute hundreds or thousands of instructions, modifying more state (i.e. memory) than a processor can buffer. To surmount this problem, TLS implementations expand the buffer of speculative state to include the processor's caches. Now the caches must be able to identify speculative state to prevent that state from becoming visible to other processors.

Prior to our work, the Cello simulator did not allow different processors to share a cache if the cache maintained speculative state. Our work extended the Cello simulator to allow multiple processors to put speculative state in the same physical cache.

3.1 Cache line state

To identify the speculative state Cello simulates a pair of extra bits for each cache line. The first bit, speculatively loaded (SL), indicates that a speculative instruction has read data in that cache line. The second bit, speculatively modified (SM), indicates that a speculative instructions has written data to that cache line.

We extended the cache line state to contain a pair of SL/SM bits for each processor that could speculatively access that line. Our current implementation divides a 32-bit word into 16 2-bit pairs. Hence we can support at most 16 processors per shared cache, but one could extend this technique to support an arbitary number of processors.

3.2 Cache line transition function

Allowing multiple processors to access the same cache line required changes to the cache line transition function. Our implementation was based on the description in Steffen *et. al.* [1].

To implement the state transition function, we maintain a *more-speculative-mask* for each processor. If processor j's more-speculative-mask contains a 1 in position i, then processor i is executing an epoch that is more speculative than processor j. The simulator updates each of these masks whenever an epoch starts or ends. These masks allow the simulator to quickly determine if a given line has been speculatively loaded and/or speculatively modified by any processors more speculative than a given processor.

For instance, consider the case where processor i, speculatively executing epoch j, writes to a line that is speculatively shared. All epochs more speculative than j that have loaded or modified that line must be violated. By comparing processor i's more-speculative-mask with the SL and SM bits the simulator can determine which epochs must be violated. By carefully laying out the masks and SL/SM bits, we were able to use bitwise operations to determine violations quickly. Any more speculative epoch is then violated. Otherwise the simulator checks if processor i has modified the line. If not, it issues an upgrade request speculative in order to get obtain exclusive access to the line. If this request fails, the line is shared and since its value must be broadcast if the epoch commits, the line is added to the epoch's ownership required buffer. And finally, the cache line is marked to note that processor i has speculative modified the line.

Other cases follow a similar format for detecting violations.

3.3 Memory system

Cello's private cache implementation groups an instruction cache, a data cache, an ownership request buffer (ORB), a notify-modify request buffer (NMRB), and a processor into a *memory-system*. In the shared cache implementation, each memory system still has one instruction and one data cache. To allow multiple processors to share a cache, we extended the memory system object as follows. First, the memory system object was modified to allow multiple processors to connect with it. Second, since the ORBs and NMRBs are designed to be associated with one processors (actually one epoch context), we modified the memory system to build one ORB and one NMRB for each processor that connected to it.

4 Benchmark Results and Discussion

4.1 Overview

We ran three benchmarks on our shared cache implementation: compress95, ijpeg, and li. Each benchmark ran with each of the following processor configurations: 1 processor, 4 processor private cache, 4 processor shared cache, 16 processor private cache, and 16 processor shared cache. For each processor configuration we used four different data cache sizes: 32KB, 64KB, 128KB, and 512KB. Note that these sizes represent *total* cache size. In the shared cache configuration, each processor had access to a single data cache of that size. However, in the private cache configuration, we divided the cache size equally among the processors. For example, consider 4 processors with 128KB of total data cache. In the shared cache configuration each of the 4 processors had access to the entire 128KB data cache, while in the private cache configuration each processor had access to its own 32KB data cache. By considering total data cache size instead of per processor data cache size, we can make comparisons about the efficiency of different ways of dividing up the data cache. All of the data caches were 2-way associative. In addition, our experiments used 32KB 4-way associative instruction caches.

Overall we found that shared caches performed better than private caches given the same total cache size. Furthermore, we found instances where the shared caches also performed better even when each private cache was the same size as the single shared cache. For example, consider the performance of the ijpeg benchmark with 16 processors as show in Table 2. The 32KB shared cache configuration performs better than the 512KB private cache configuration (377.1 vs 436.7 million processor cycles) despite the fact that each processor in the private cache has its own 32KB cache. In this case the shared cache configuration had one sixteenth the total cache size as the private cache, yet performed 14% faster than the private cache configuration.

With TLS-enabled benchmarks the processors sharing a cache exhibit cooperative rather than competitive memory access patterns in the benchmarks tested. As the processors execute different iterations of the same loop, they tend to access similar memory locations. This result is encouraging, since one might as well have expected a more competitive scenario, in which processors compete with each other, evict each other's data, and cause epoch violations in speculative threads.

4.2 compress95

In the case of compress95, the shared cache configurations produced speedups of up to 20%, and the best performing private cache configuration was slower than each of the shared cache systems (Table 1 and Figure 1 left). For the single processor setup we observed a a sharp speedup and a dramatic reduction in the number of misses when the cache size increased from 128KB to 512KB. We suspect that a 512KB cache was required to hold the benchmark's entire working set. When we increase the number of processors to 4 and then to 16, the sharp increase in performance disappears for the private cache systems, since each processor now has a smaller cache. We observed steady gains in performance as the cache size increases. With a shared cache, however, the execution time is shorter for all cache sizes, and the jump from 128KB to 512KB cache produces the same sharp drop in execution time and miss count that we observed in the single processor

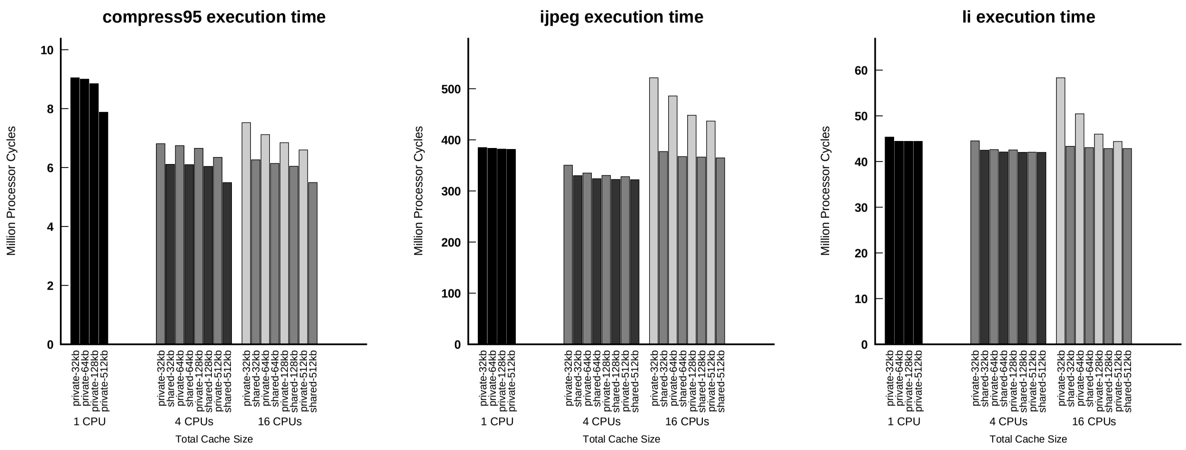

Figure 1: Number of processor cycles performed

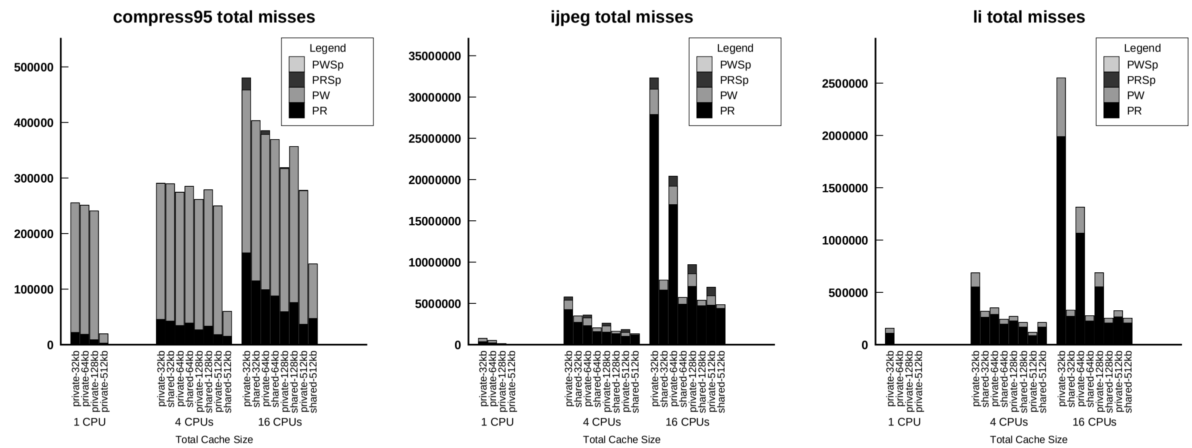

Figure 2: Number of Total Misses (summed over all data caches in the memory systems)

configurations. Presumably, the processors in the shared cache configuration are able to share data and avoid thrashing. Given a large enough cache, considerable performance gains are achieved with a shared cache, which are not obtained with private caches of equivalent total size.

num processors	1	4		16	
cache size (kb)	-	private	shared	private	shared
32	9.047	6.810	6.109	7.524	6.239
64	9.001	6.745	6.099	7.118	6.139
128	8.848	6.653	6.039	6.843	6.044
512	7.878	6.346	5.490	6.598	5.491

Table 1: compress95 execution time (M proc sycles)

4.3 ijpeg

Results from the ijpeg benchmark are similar to the compress95 data. ijpeg, which contains parallelized loops of four iterations, performs considerably worse with 16 processors and private caches because data strides across the caches as the four loop iterations are executed by alternating teams of 4 processors (Table 2 and Figure 1 center). With a shared cache, we are able to hide much of that latency: the processor read miss count (PR) is cut by more than 75%, while the speculative misses are completely eliminated. It is likely that cooperative data access patterns are responsible for the improvement; configurations with the same per-processor cache sizes (512KB 16CPU private and 32KB 16CPU shared) yield very similar miss counts, despite the large difference in total cache size.

num processors	1	4		16	
cache size (kb)	-	private	shared	private	shared
32	384.9	350.4	329.9	521.4	377.1
64	383.4	335.0	324.0	485.8	367.2
128	381.9	330.4	322.5	448.0	366.2
512	381.4	327.9	321.9	436.7	364.7

Table 2: ijpeg execution time (M proc cycles)

4.4 li

The li benchmark did not seem to parallelize successfully, as the execution time improvement was minimal for 4 and 16 processors (Table 3 and Figure 1 right). The shared cache systems performed equally well at all cache sizes, and had comparable execution times on the 4 and the 16 processor systems. Private caches of 8KB or more (32KB 4CPU and 128KB 16CPU systems) also showed similar performance, but smaller caches accessible by each processor resulted in a large number of misses (Figure 2 right). Shared cache configurations eliminated this effect; performance was comparable for all configurations, as long as per-processor cache size was at least 8KB.

5 Conclusions and Future Work

We have shown that shared caches can offer a performance improvement within the TLS paradigm. The shared cache memory configurations outperformed the private cache memory configurations, and often afforded great savings in cache size. The execution times of all the benchmarks on the shared cache configurations were consistently shorter, and there were fewer cache misses in almost all cases.

num processors	1	4		16	
cache size (kb)	-	private	shared	private	shared
32	45.35	44.52	42.48	58.33	43.33
64	44.43	42.61	42.09	50.44	43.05
128	44.42	42.54	42.01	45.99	42.84
512	44.42	42.05	42.00	44.39	42.84

Table 3: li execution time (M proc cycles)

All the benchmarks performed better with 4 processors than with 16 processors. With 16 processors the shared cache configurations brought the performance much closer to that of 4 processors. However, in order to justify the use of 16 processors, other ways of increasing the parallelism in the program must be found, as the performance benefits of shared caches are not sufficient to reverse this trend.

Our basic conclusion is that shared cache systems exploit cooperative data access patterns to improve performace, while competitive behavior does not make as significant an impact in the benchmarks analyzed. However, our results are preliminary and further investigation is required. Cache performance should be evaluated on a wider variety of benchmarks and memory system configurations. (Preliminary tests using greater associativity on the larger caches showed similar relative performance among the memory configurations.) Also, it would be interesting to investigate the epoch violation patterns in shared caches as compared with private caches.

Several other techniques could offer further performance gains. First, we did not implement a cache coherence scheme, and hence limited the shared cache configurations to a single data cache. Extending the implementation to handle coherence should not present a significant challenge. We already have implemented a proper invalidation of more speculative epochs within the a single cache. Much of this code can be reused to implement the cache coherence scheme.

Second, one might include the support of multiple writers. Multiple writer support allows different processors to write to different words within a cache line without invalidating the more speculative epoch. For instance, in a case in which each iteration of a loop updates consecutive positions in an array, if more than one item fits within a single cache line, a less speculative epoch may unnecessarily violate a more speculative epoch when writing its item. Multiple writer still maintains the restriction that only one speculative epoch can modify a single cache line. The cache maintains fine grain bits for each line that indicate which words within the line have been modified. When a second epoch tries to write to the line, instead of invalidating that epoch, the line is copied to another cache line within the same associative set and the approiate fine grain bits are set. When one of these epochs commits only the modified words need to be broadcast (if the line was in a shared state.)

References

[1] J. G. Steffan, C. B. Colohan, and T. C. Mowry. Extending Cache Coherence to Support Thread-Level Data Speculation on a Single Chip and Beyond. December 1998.

[2] B. A. Nayfeh and K. Olukotun. Exploring the Design Space for a SharedCache Multiprocessor. In Proceedings of ISCA-21st, pages 166–175, Chicago, Illinois, Apr. 1994. IEEE Computer Society Press.