An Analysis of Cache Sharing in Chip Multiprocessors

Brian Forney Steven Hart Matt McCormick

Final Project Report
CS 757: Parallel Computer Architecture

Computer Sciences Department
University of Wisconsin–Madison

Abstract

We present the effects of L1 and L2 cache sharing on cache miss rates, cache line invalidations, and constuctive and destructive interference. The most important finding of this paper is that a system configuration that shares L2 caches, does not share L1 caches, and does not enforce inclusion between the L1 and L2 caches will produce the highest performance cache and communication hierarchy for a chip multiprocessor. This is due to the relatively high speed of communication through the L2 cache but the low effects of L2 sharing on L1 performance — if inclusion is not enforced. Sharing at the L1 level produces too many conflict misses at this all important resource.

1 Introduction

Advances in integrated circuit processing have opened the door to multiple processor cores on a single chip, or chip multiprocessors (CMP) [2, 7]. CMPs enable the sharing of caches, whereas previously, sharing was prohibitive due to high latencies and narrow bandwidth of off-chip communication. Our work analyzes the effects of sharing L1 and L2 data caches on an eight processor CMP.

Sharing of caches in CMPs has several advantages. First, processors may induce *constructive interference*. Constructive interference occurs when one processor loads data into a shared cache which is later used by other processors sharing the same cache. Cache misses are avoided when constructive interference occurs. Second, communication traffic can be more evenly distributed between communication links across a system. In a system without shared caches, all communication must traverse a common bus or network. However, with shared caches, communication can occur closer to the processors. The common interconnect in shared cache systems scales better and can be engineered to have lower bandwidths decreasing costs. Third, the total size of caches and thus chip space can be reduced. However, this depends on the amount of constructive interference.

While sharing of caches in CMPs may improve performance and scalability of systems, several effects may diminish the advantages of cache sharing CMPs. Constructive interference's antipode, *destructive interference*, increases the number of misses. The increase is due to cache conflicts between cache sharing processors. Second, sharing caches adds to the average memory access latency due to arbitration between multiple processors at the L1 cache level and multiple L1 caches at the L2 cache level.

Which effects dominate are dependent upon the workload, cache sharing hierarchy, cache characterisitics such as associativity, and processor instruction fetch and data load and store strategies. In this paper, we explore how workload, cache sharing hierarchy, and associativity affect cache miss, L1 and L2 invalidations, and constructive and destructive interference using a subset of the SPLASH-2 benchmarks [8].

The remainder of the paper is organized as follows. Section 2 discusses the simulation environment. Results are presented in section 3. Section **??** contains related work. We conclude in section 4.

2 Simulator

SimpleScalar MP [6] is a modification of the the SimpleScalar [3] simulator suite. The multiprocessor version includes modified versions of the fast, functional simulator (sim-fast) and the functional cache simulator (sim-cache). We modified the multiprocessor version of sim-cache (sim-mpcache).

2.1 Sharing Caches

The most important modification to sim-mpcache was to allow it to share caches in various configurations. The original sim-mpache works by adding an additional dimension to each of the internal data structures. For example, sim-cache has a simple array that represents the cache tags. sim-mpcache adds an addition dimension to the array of cache tags. The two-dimensional array holds the cache tags for each of the processors in the multiprocessor simulation. Each thread in a the multiprocessor simulator has a processor id (*pid*). Processor ids run from zero to the configured number of processors minus one. Pids are used by the simulator to index into the simulator's internal data structures.

Our modifications add an additional level of indirection to *pids*. We map each *pid* to two effective pids (*epid*): one *epid* maps a processor to an L1 cache and the second *epid* maps a processor to an L2 cache. We can configure the degree of cache sharing by mapping multiple pids to a single *epid*.

The original simulator maintained inclusion. An additional modification to the simulator was required to maintain inclusion in the shared cache simulator. If a line in an L2 cache was replaced, then that line was invalidated in the L1 cache. For the one-to-one mapping from an L2 cache back to an L1 cache maintaining inclusion was simple because both caches shared a single *pid*. In the shared cache case, if a line is replaced in a shared L2 cache, then that line must be replaced in all of the L1 caches that share the L2. For efficiency, we maintain a one-to-many reverse mapping from an L2 cache to all of the L1 caches that share it. To maintain inclusion on a cache line replacement, we simply iterate over the reverse mapping for the L2 cache and invalidate the copies of the cache line in each of the L1 caches.

Sharing caches means that we have to maintain coherence among all of the L1 caches that share a single L2. If a write modifies a cache line in an L1, then corresponding line in the L2 cache is set to MODIFIED_ABOVE state. That line has to be invalidated in all of the other L2 caches in the system. However, now that multiple L1s share a single L2, we need to invalidate any copies of the line in the other L1 caches that share this L2. We use the reverse mapping technique described previously.

2.2 Set Associativity

SimpleScalarMP models only direct-mapped caches. We believed that set-associativity was important for our study. In the original version of sim-mpcache, cache tags and state are implemented as two, two-dimensional parallel arrays indexed by the *pid* and the address index. In our implementation, we use the *epid* to index into an array of caches. An entry in this array points to an array of ways. Within each way are the cache tags and states.

2.3 Cache Misses

Cache misses is one of the metrics we used to evaluate different cache organizations. We modified the simulator to partition cache misses into their sources. The most interesting problem that we addressed was determining compulsory misses. Each address accessed by the program uses a bit vector to track references to the address for each processor. When an address is referenced, it is used to index into the the table of bit vectors. The appropriate bit vector matching the address tag is found, and the bit corresponding to the processor id is checked. If the bit is unset, then this is the first reference to this address from the current processor and this miss is accounted for as a compulsory miss.

2.4 Prefetching and Interference

In order to measure constructive prefetching, we track which processor fetched a particular cache line. When when a cache line is accessed by a processor, a data structure associated with the cache line is checked for the id of the processor responsible for fetching the cache line. If the id of the requesting processor and the fetching processor are different, we consider this a cooperative prefetch.

2.5 Simulator Limitations

The the multiprocessor version of SimpleScalar makes only the simple, functional simulators available. Consequently, it is impossible to make any conclusions about the effect shared caches have on the runtime performance of the benchmarks. We confine ourselves to studying the frequency of relevant memory system events. It is also worthwhile to note that the SimpleScalar cache simulator does not simulate instruction caches. Only the data reference stream is simulated. While this is not a major concern for L1 caches, unified L2 caches may appear to be underutilized since instructions are not occupying any lines in the cache.

3 Results

3.1 Methodology

The main focus of our research was to study the effects of processors sharing various levels of the cache hierarchy on cache miss rates, cache line invalidations, and the interference patterns between different processors accessing the same cache. Before proceeding with our experimental set-up, we will introduce some of the notation used through-out the rest of this paper. In particular, cache sharing configurations are documented in the following manner:

num processors:num L1 caches they map to / num L1 caches:num L2 caches they map to

The simplest way to illustrate our notation for the different cache sharing configurations is with an example. A description such as 2:1/4:1 means there are 2 processors sharing a single L1 cache and 4 L1 caches sharing an L2 cache. In the case of an 8 processor system this means there are 4 L1 caches and a single L2 cache.

Our experiments were run using 6 different benchmarks. These included Barnes-Hutt, fft, ocean-contig, ocean-noncontig, water-nsquared, and water-spatial. A brief description of of these benchmarks is provided in the next section. For a more complete description, refer to [8]. We studied for 10 different cache sharing configurations. These are: 1:1/1:1, 1:1/2:1, 1:1/4:1, 1:1/8:1, 2:1/1:1, 2:1/2:1, 2:1/4:1, 4:1/1:1, 4:1/2:1, 8:1/1:1. Lastly, we vary the set-associativity of the L1 and L2 caches in the following way: both direct mapped, L1 direct mapped / L2 4-way set-associative, L1 2-way set-associative / L2 4-way set associative.

We fix the L1 and L2 cache line sizes at 64 bytes, the L1 cache size at 64 KB, and the L2 cache size at 1 MB. Regardless of the number of processors sharing a cache, the size of that cache does not change for almost all of our data. The reason for this is that the effects of varying cache size are very well studied and understood. This also helps us prune the number of experiments down to a reasonable number to study (there are still 180 different tests that needed to be run) and helps us focus on what we care to study — miss rate, invalidations, and interference. In a few cases, graphs with different cache sizes are presented as they help to illustrate a specific observation.

3.2 Benchmarks

A brief description of each of the benchmarks we used in our study is provided to help make it clear why some of the results appear as they do. The information for this section was derived from [8]. Barnes-Hutt and both water benchmarks tend to access rather small amounts of data as compared to FFT and the two ocean benchmarks. Barnes-Hutt and ocean produce considerably less remote traffic for 8 processors than fft and the water benchmarks. The water benchmarks do considerably more synchronization than any of the other four benchmarks.

3.3 Miss rate

3.3.1 L1 Miss Analysis

Before beginning the analysis of the cache miss rates, it is important to make a couple of points. First, we consider a coherence miss to be any cache miss to a line that would have been valid had the system only had one processor, one L1 data cache, and one L2 cache. This invalidation may not necessarily be the result of a processor obtaining Read Exclusive permission to a block. It could also occur in a system that enforces inclusion and where two L1 caches are sharing an L2 cache. If one of the L1 caches takes a miss on an address and the L2 also misses on that address, then the line must be brought into the L2. If the line being replaced happens to be cached in another L1 that is sharing this L2, that L1 must invalidate its copy even if no other cache has written to it. We consider this to be a coherence miss.

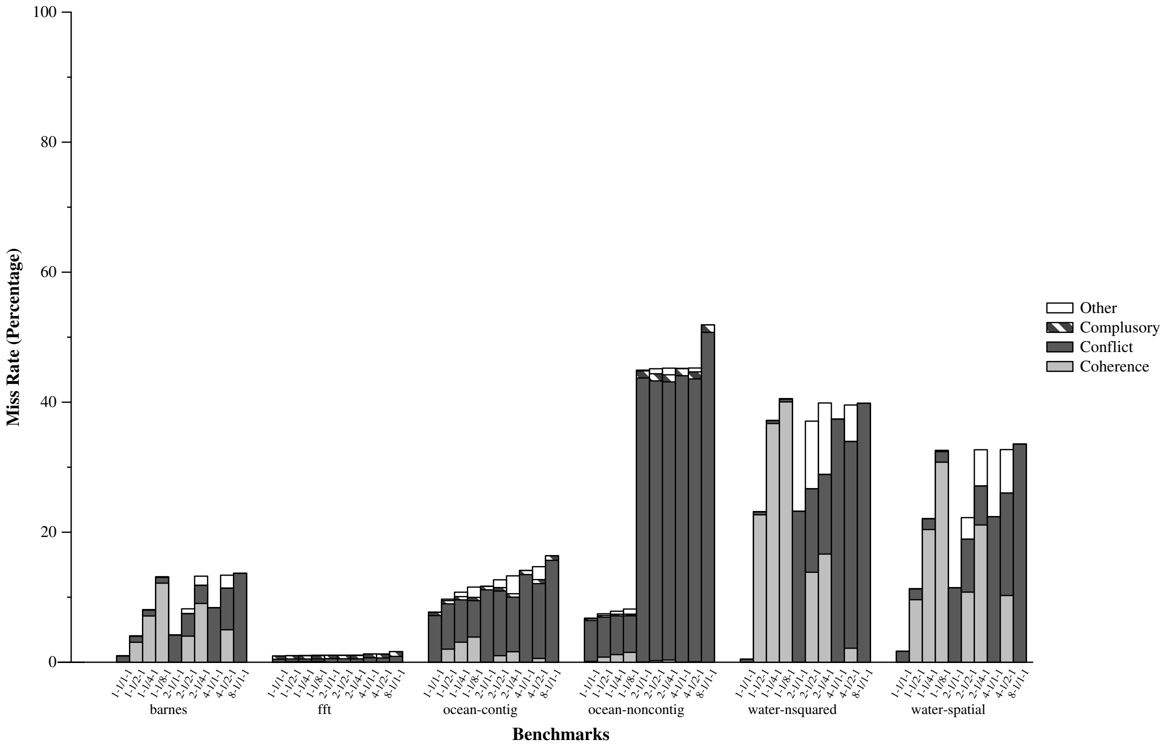

Figure 1: **L1 miss rate using direct mapped L1 and L2.** Individual L2 cache size remains fixed as the number of processors sharing that L2 increases.

The reader will also notice one of the categories for a cache miss is entitled *other*. This is any cache miss to a line that has been referenced in the past, does not match any of the cache tags, and finds no sets with an invalid entry in the particular cache line to which it maps. On first glance this would appear to be a capacity miss since an infinite cache would contain all past references. However, if the cache were infinite, the line may have been invalidated due to coherence at some point so it is not possible to guarantee these are all capacity only misses. Hence we label these as *other misses*.

Our study of the effects of cache sharing on L1 and L2 miss ratios show us several very interesting results. We begin with an analysis of the L1 cache miss ratios. Perhaps one of the most interesting results is that cache sharing can mean that cache size and set-associativity can have an impact not only on capacity and conflict misses, but can also effect coherence misses in a cache hierarchy that forces inclusion. This is in opposition to the findings of Barroso, et al. [1]. Here it was stated that as cache sizes increase, coherence misses in multi-processor systems will begin to dominate because they remain unaffected by cache size. However, consider Figures 1 and 2. The experiments represented by

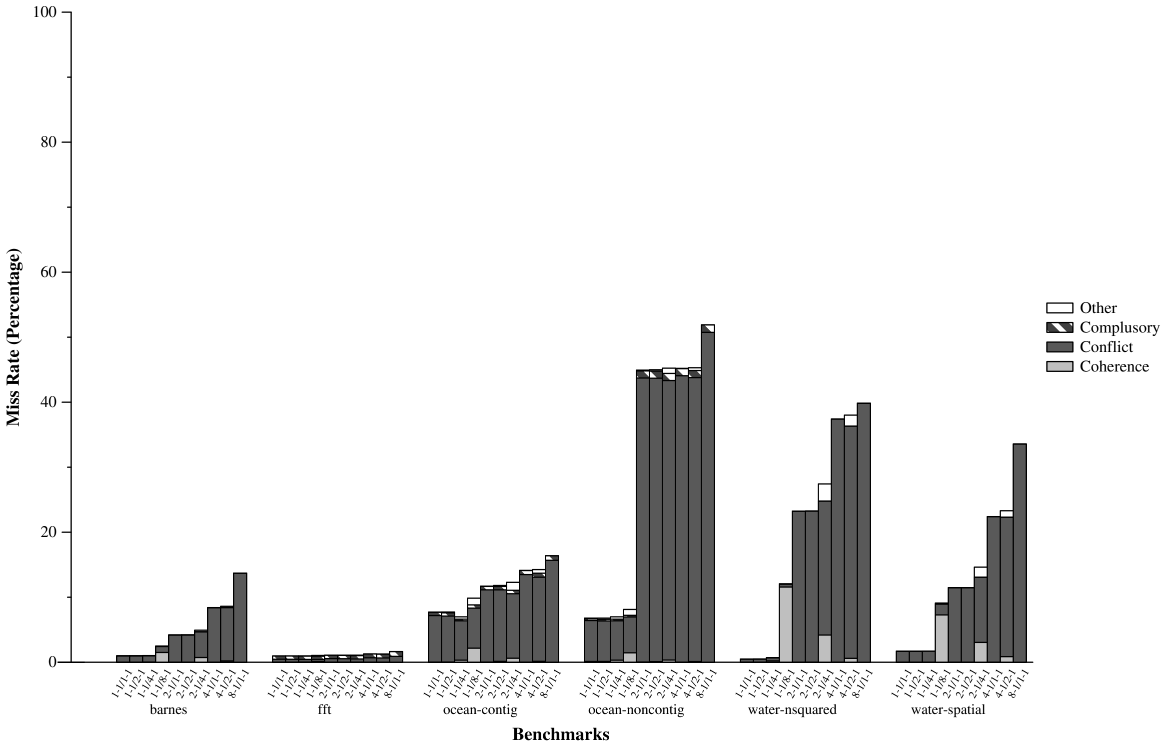

Figure 2: **L1 miss rate using direct mapped L1 and 4-way L2.** Individual L2 cache size remains fixed as the number of processors sharing that L2 increases.

these two graphs differ only in the degree of L2 cache set-associativity. The Barnes-Hutt, water-nsquared, and water-spatial benchmarks show drastic decreases in the coherence misses and overall misses for all of the shared L2 cache only configurations. Figure 4 shows these 3 benchmarks run with a direct mapped L2 cache that is identical to that of Figure 1 except it has a cache size that increases proportionally with the number of caches sharing it. Again we see the coherence misses reduced drastically when compared to Figure 1.

The explanation for this phenomenon is quite simple. It is a result of forcing inclusion between the L1 and L2 caches. If an L2 cache line is evicted for any reason, it must be evicted in all of the L1 caches that are sharing that L2. As more L1 caches share an L2 cache, the number of L2 conflicts, and hence L2 replacements and L1 coherence invalidations, will increase unless additional steps are taken. These steps include higher set-associativity, a larger L2 cache size, or creating a cache hierarchy that does not force inclusion. This latter step is left as future work and will not be examined in this paper.

Our experiments also helped to reinforce some commonly held beliefs in regards to caches. We see L1 cache

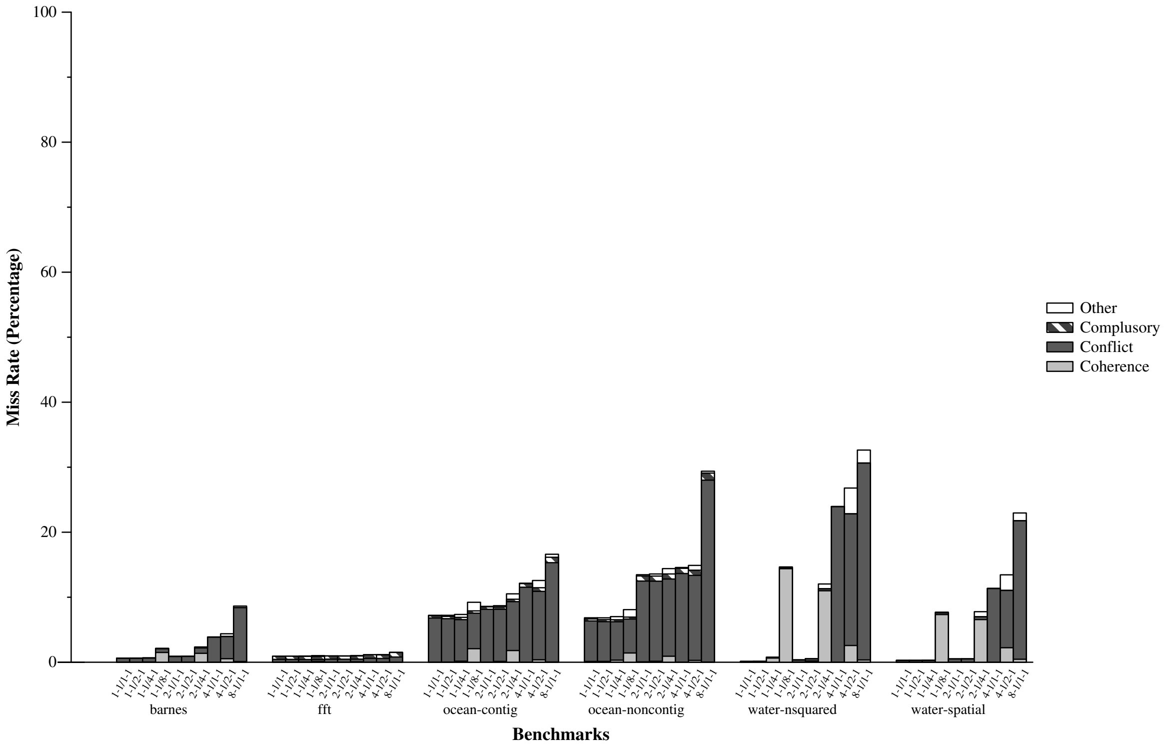

Figure 3: **L1 miss rate using 2-way L1 and 4-way L2.** Individual L2 cache size remains fixed as the number of processors sharing that L2 increases.

misses increase drastically as the degree of L1 sharing increases (Figures 1, 2, and 3). This is again due to the fact that we did not increase the L1 cache size as the amount of sharing increased. This effectively leads to a smaller cache because the number of references to this cache increases in proportion to the number of processors connected to it. Another well understood phenomenon is that of increased set-associativity. As one would expect, higher L1 set-associativity leads to lower miss rates (Figure 2 vs. Figure 3) when sharing L1 caches. This is because multiple processors are frequently presenting conflicting data to the cache at the same time. The graphs show this in that the vast majority of the misses for these two graphs — when sharing L1 caches — are due to conflict misses, and higher set-associativity drastically reduces the overall effects of conflicts.

An interesting side note is that when no L1 sharing is going on (regardless of the amount of L2 sharing), L1 set-associativity has little or no effect on cache misses due to conflicts. This shows that the benchmarks here have been very well written and reduce the amount of data conflicts a single processor produces.

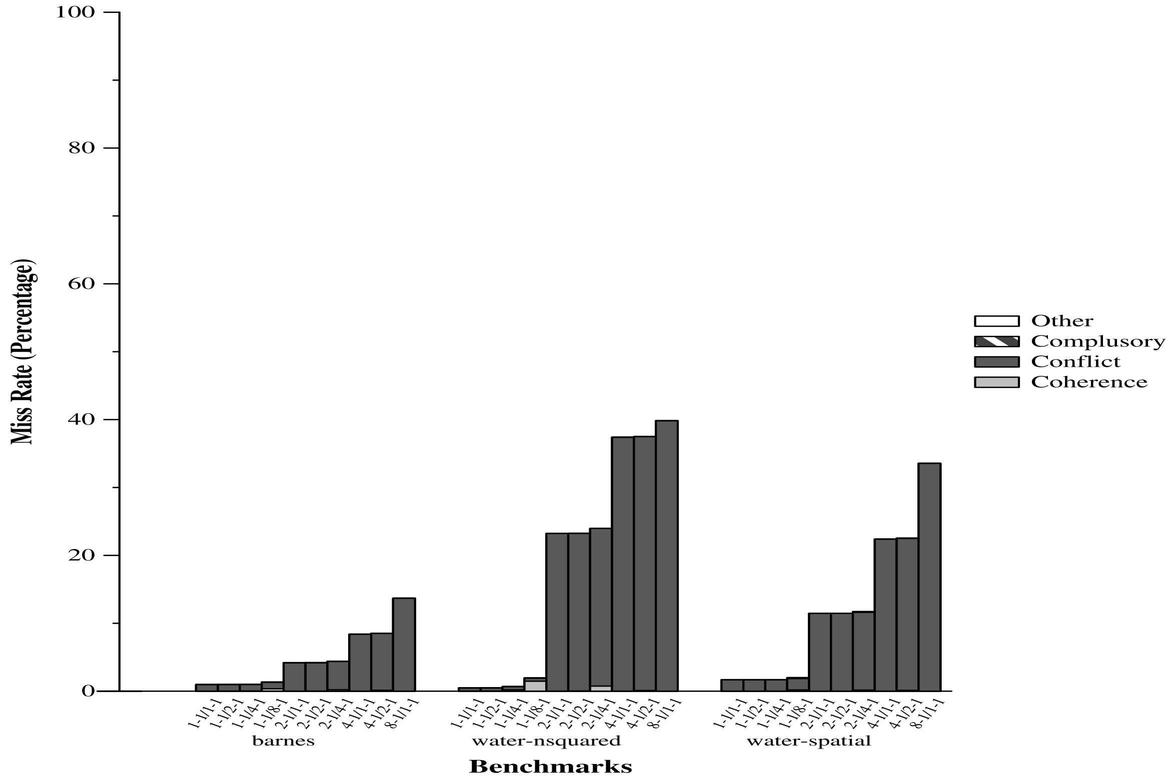

Figure 4: **L1 miss rate using direct direct mapped L1 and L2 with 8 MB total L2 caches.** Individual L2 cache size increases as the number of processors sharing that L2 increases.

3.3.2 L2 Miss Analysis

The effects of cache sharing on miss ratios in the L2 cache are both more complicated and more interesting. Figure 5 shows us that most of the benchmarks behave similarly to those for the all direct-mapped L1 caches: an increase in the number of L2 misses for more processors mapping to a single L2 cache. What makes this interesting is the ocean-noncontig benchmark. This shows us that if no L1 sharing is going on, we see the expected results — an increase in L2 misses as more processors map to an individual L2. However, once we begin to share L1 caches, the L2 miss rate drops drastically. We believe this is due to the effects of destructive interference in the L1 cache. Figure 1 shows ocean-noncontig conflict misses skyrocketing for shared L1 configurations. This would mean that there are a significantly higher number of accesses to the L2 cache. As long as the data is not getting removed from the L2 cache due to conflicts, there should be many more hits and about the same number of misses in the L2 cache. Our raw data (not presented in this paper) shows that for no L1 sharing there are approximately 9 million L2 accesses and about 3 million L2 misses. If the number of L1 caches is reduced to two (4 processors sharing an L1), the total number of

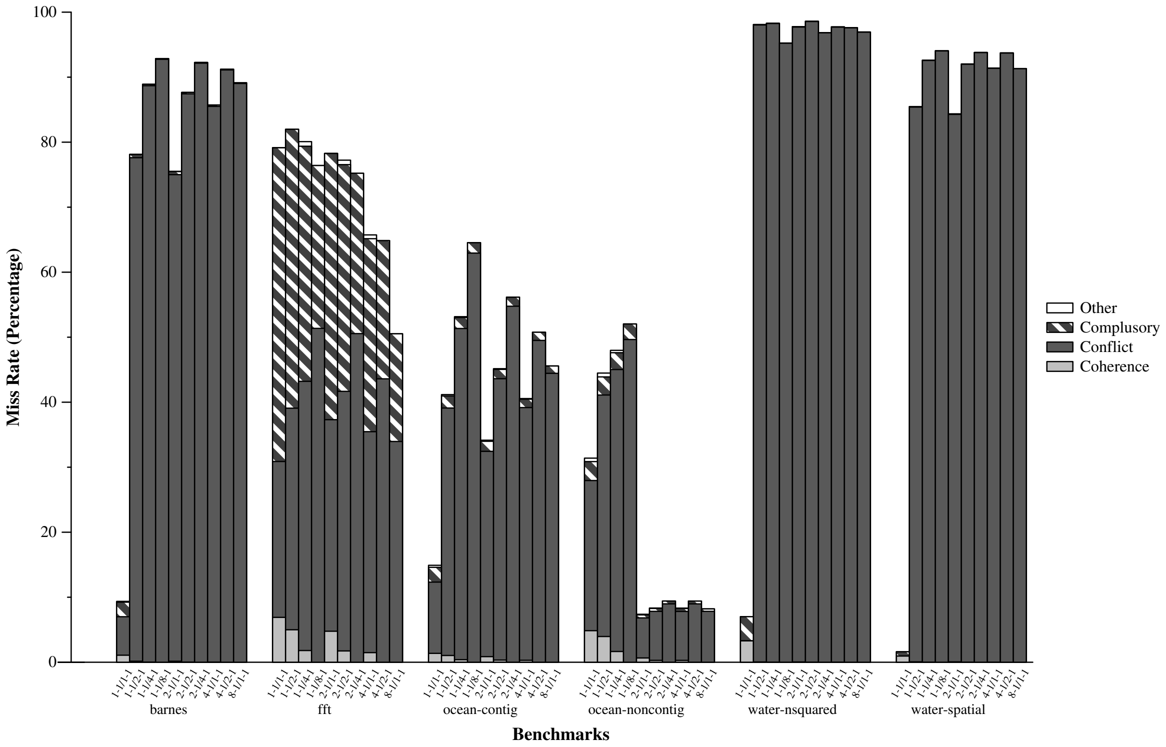

Figure 5: **L2 miss rate using direct mapped L1 and L2.** Individual L2 cache size remains fixed as the number of processors sharing that L2 increases.

references to the L2 cache goes up to approximately 60 million with the number of misses remaining relatively low at around 5 million!

Another interesting result of the experiments are the effects of set-associativity on L2 *conflict misses*. Figure 7 shows that for Barnes-Hutt, fft, Ocean-contig, and both water benchmarks the worst sharing configurations for the L2 miss ratio were 1:1/8:1, 2:1/4:1, 4:1/2:1, and 1:8/1:1. The reader should note that all of these configurations lead to all 8 processors mapping to a single L2 cache. We have concluded that the reason for this is because the critical data working set of each of these benchmarks is no longer fitting in the L2 cache when 8 processors are mapping to a single L2. To prove this point, we can examine Figure 8. We see here that as the cache size increases in proportion to the amount of sharing we see the L2 miss ratio for all of these mappings falls into line with the other configurations.

Perhaps the most startling results are the incredibly high L2 miss rates for most of the benchmarks when working with a direct-mapped L1 and L2 cache (Figure 5). Barnes-Hutt, ocean-contig, and both water benchmarks show most of the misses to be a result of conflict misses. We believe it is because these programs are written in such a way

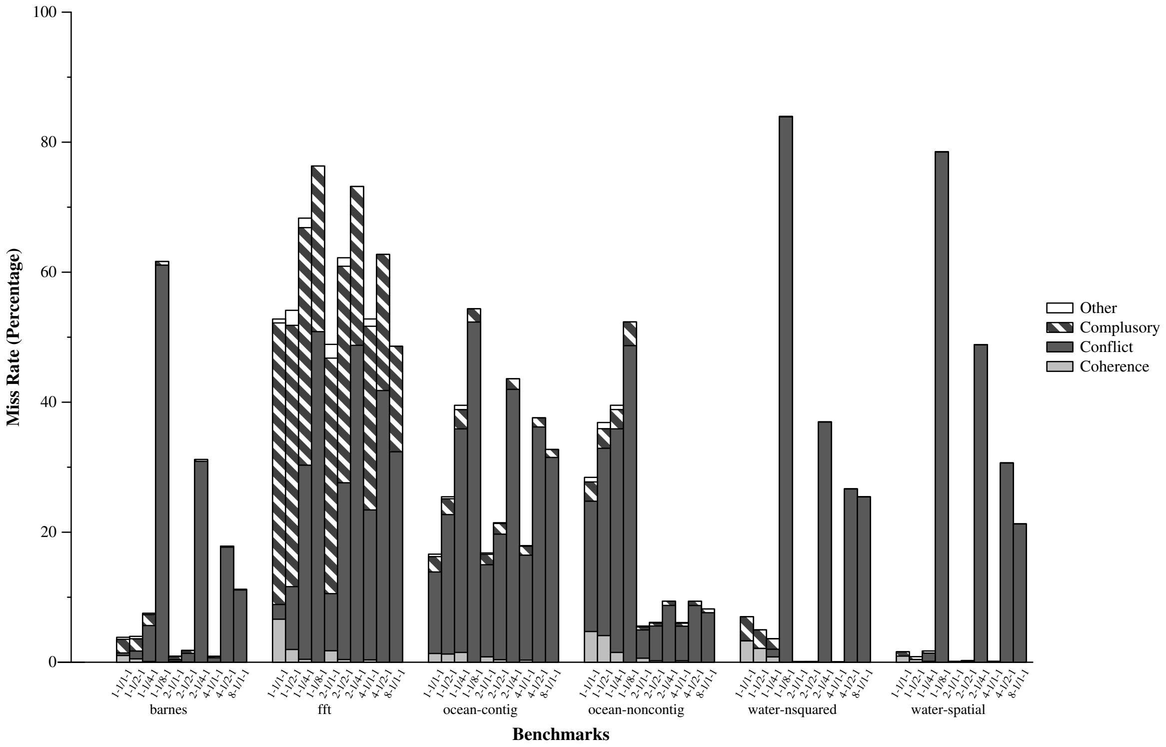

Figure 6: **L2 miss rate using direct mapped L1 and 4-way L2.** Individual L2 cache size remains fixed as the number of processors sharing that L2 increases.

that many of the processors are accessing data at different addresses that map to the same location in the L2 and not because there is not enough room in the L2 to hold the data (except for those discussed in the previous paragraph). As discussed above, once associativity is added the conflict misses almost disappear for some of the configurations. If the problem was capacity, we would not see this sharp drop in conflict misses. If these mapping conflicts do exist in the application's code, this could severely hamper performance and would point to the importance of rethinking the way some parallel applications are written if they are to share an L2 cache.

Lastly, we would like to discuss the fft and the ocean benchmarks. Consider Figures 5 and 7. These benchmarks do receive some overall benefit from increased L2 set-associativity, but it is no where near the benefit received by most of the other benchmarks. The fft benchmark in particular has a large number of *coherence misses*. This makes sense if we consider the L1 miss rates for fft are extremely low. This means a large portion of the references to the L2 cache are for data that has never been accessed. No cache configuration will reduce these misses.

More interesting are the results of increasing the size of an individual L2 cache for each of the 3 benchmarks

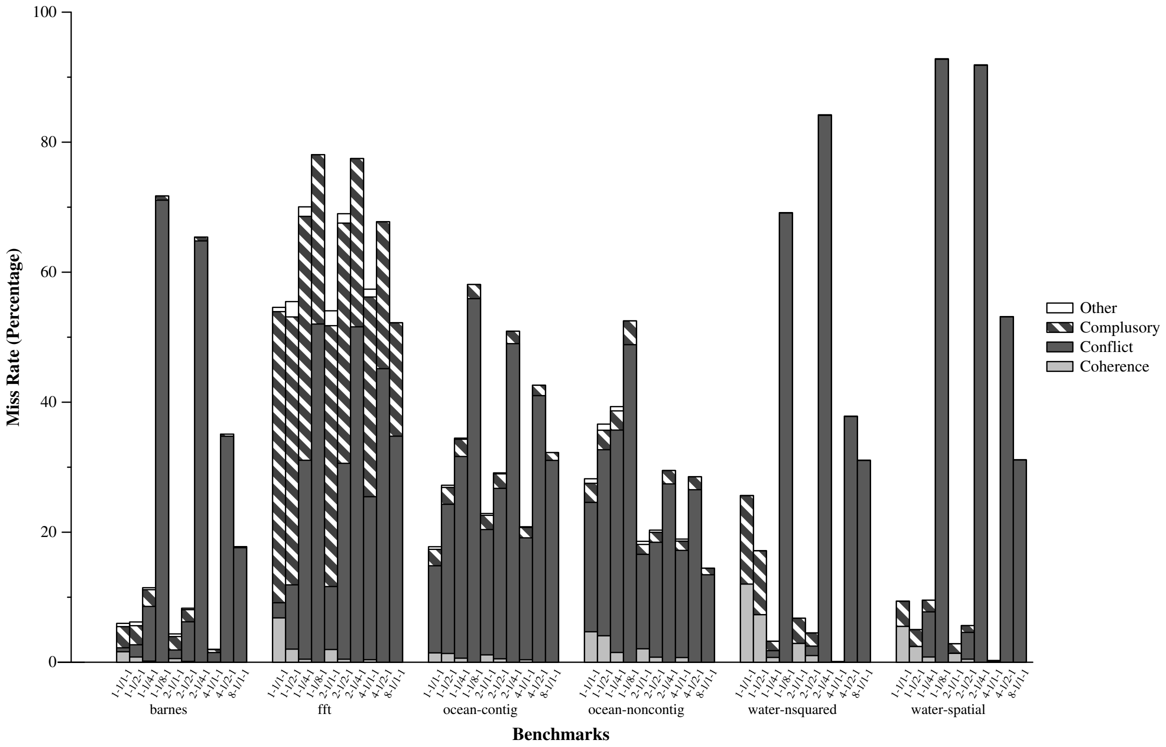

Figure 7: **L2 miss rate using 2-way L1 and 4-way L2.** Individual L2 cache size remains fixed as the number of processors sharing that L2 increases.

in question. As the amount of sharing of that cache increases, it can be seen that for a constant L2 size(Figure 7) the conflict misses increase as the amount of L2 sharing increases. However, in the case of increasing L2 cache size (Figure 8) we see decreasing conflict misses. We believe the reason for this is because each processor for fft and the ocean benchmarks are accessing much of the same data — but this data does not fit into a single L2 cache. Hence, for non-increasing L2 size and increased L2 sharing, the conflicts in the L2 rise because it gets increasing difficult to fit the working set into the L2 cache. However, if we allow the L2 cache to grow, not only does more of the working set fit in the L2 cache, but the processors begin prefetching data for one another. Both of these effects result in a lower cache miss ratio.

3.4 Invalidations

We measured the number of invalidations that occurred in the L1 and L2 caches across the different benchmarks and cache organizations. Figure 9 shows the number of invalidations for the L1 cache, where both the L1 and L2 caches are

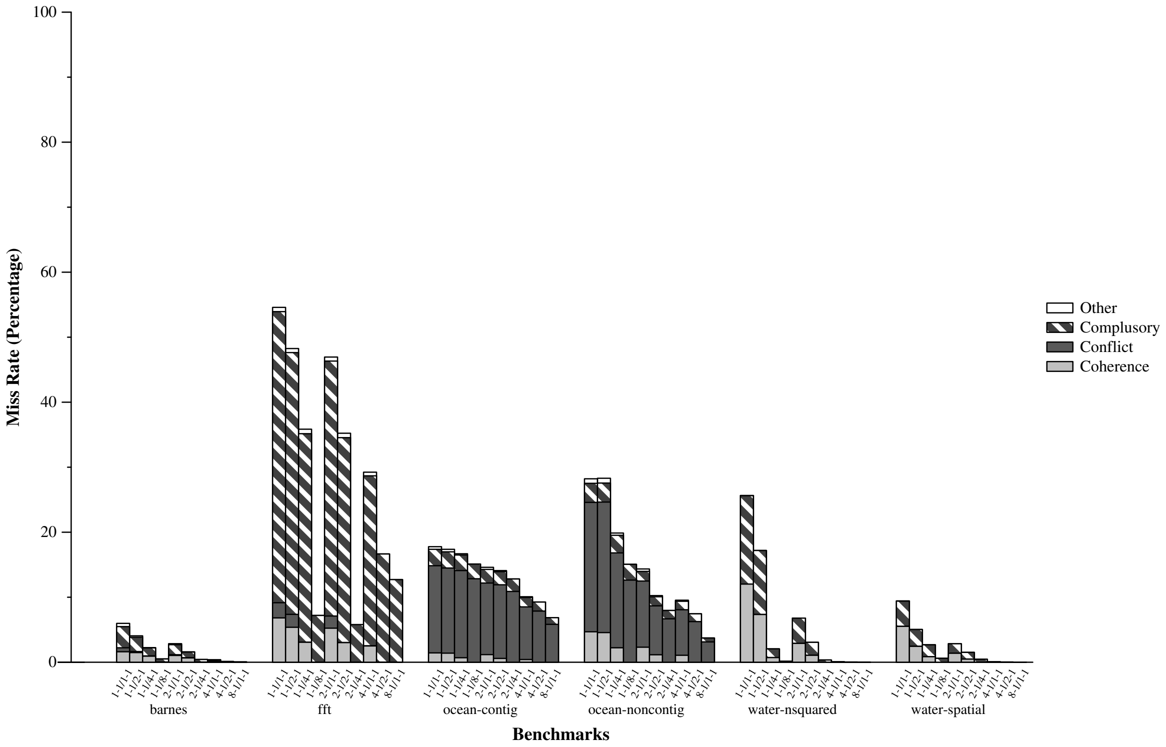

Figure 8: **L2 miss rate using 2-way L1 and 4-way L2 with 8 MB total L2 caches.** Individual L2 cache size increases as the number of processors sharing that L2 increases.

direct mapped. Clearly, fft is the most well-behaved of all the benchmarks. It has the fewest number of invalidations over across all of the cache configurations.

3.4.1 Direct-Mapped Caches

Table 1 shows the number of invalidations normalized to each benchmark's unshared configuration. The first three columns of Table 1 show the effect of increasing the degree of L2 cache sharing. Clearly, the result is an increase in the number of invalidations. When two L1 caches share an L2 cache, we see a dramatic increase in the in the number of invalidations in the water benchmarks. Barnes-Hutt and ocean-contig show less dramatic, but substantial, increases in invalidations. Moderate increases are seen in both fft and ocean-noncontig. Many of the benchmarks show a doubling of the number of invalidations as the number of L1 caches sharing a single L2 cache moves from two to four.

It is important to remember that as we increased the degree of sharing among caches, we did not increase the size of the shared caches. As more L1 caches share an L2, the amount of L2 cache per L1 cache decreases. We believe that this explains the dramatic increase in L1 invalidations observed when the degree of sharing is increased. Maintaining

Benchmark	Normalized Invalidations								
	1-1/2-1	1-1/4-1	1-1/8-1	2-1/1-1	2-1/2-1	2-1/4-1	4-1/1-1	4-1/1-2	8-1/1-1
barnes	1357.09	3102.95	5282.46	0.47	2009.63	4035.93	0.16	2068.94	0.00
fft	10.34	12.24	15.13	0.51	5.37	7.14	0.25	3.61	0.00
ocean-contig	50.15	69.78	95.31	0.89	48.64	72.04	0.36	42.90	0.00
ocean-noncontig	5.71	6.99	8.13	0.69	11.05	11.58	0.24	7.06	0.00
water-nsquared	2523.27	4061.49	4291.41	0.55	2575.96	2839.32	0.26	1113.25	0.00
water-spatial	4334.23	9173.54	13736.61	0.42	6804.93	11045.00	0.07	6211.70	0.00

Table 1: **Normalized L1 Cache Invalidations**

inclusion causes more invalidations as the degree of L2 sharing is increased. In Section 2 we said that the simulator enforced inclusion, and one issues in modifying the simulator was the a cache line replaced in a shared L2 could potentially invalidate cache lines in all of the L1 caches that are sharing the L2. Imagine an 8-processor configuration that does no cache sharing. Suppose that the cache has a copy of the a cache line L in the shared state. If one processor replaces that line in its L2 cache, that will be invalidated on *only* its L1 cache. Now imagine the same eight-processor configuration, but with a single, shared L2 cache. In this case, when L is evicted from the cache, L can be evicted from *every* L1 cache in the system. In the eight-processor shared system there is the potential to incur *eight times* the number of L1 cache invalidations when a single line in the L2 is evicted.

In Table 1, we see some interesting behavior that occures when L1 caches are shared. When two processors share an L1 cache and each L1 cache has its own L2, the number of invalidations is less than the number of invalidations in the unshared base case. However, the number of L1 invalidations increases substantially when L2 caches are shared. This trend seems to be the results invalidations due to inclusion, discussed in the paragraph above.

3.4.2 Set-Associative Caches

Figure 10 shows the number of invalidations for a simulated four-way set-associative L2 cache. The most notable feature of this graph is that the shape of the graph does not change much from the direct-mapped base case. However, as we will see below, the number of invalidations does change when the associativity of the L2 cache is increased.

Increasing the associativity of the the L2 cache has a significant effect on the number of invalidations. Table 2

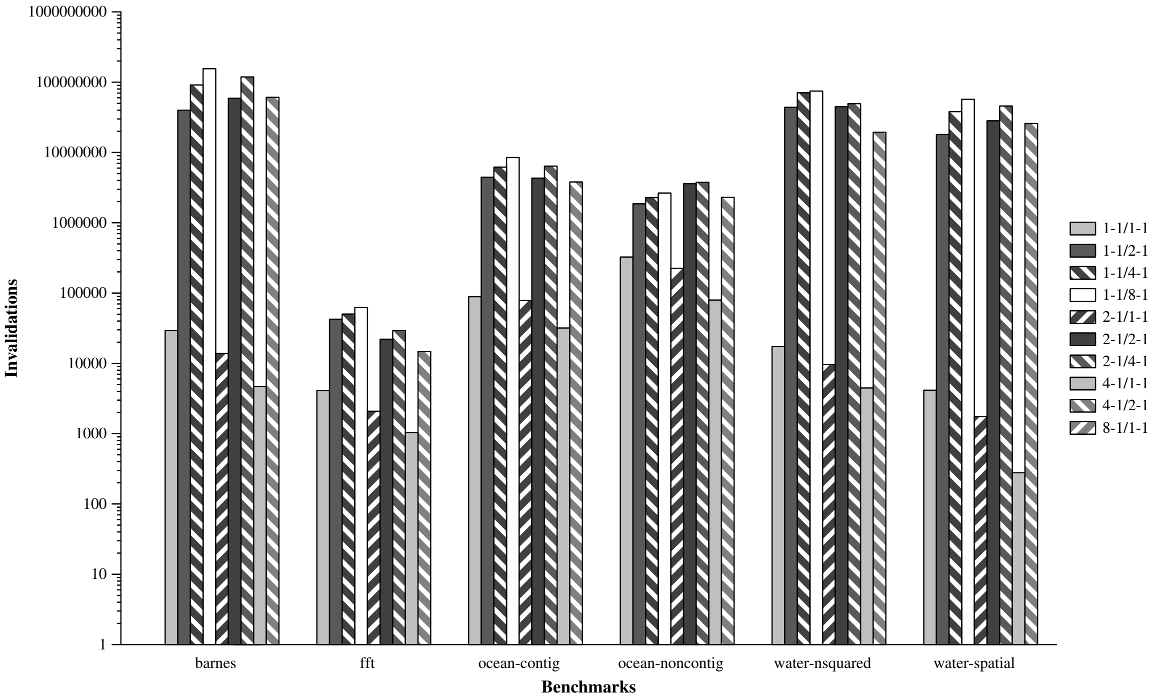

Figure 9: **L1 invalidations using direct mapped L1 and L2.**

shows the change in the number of invalidations when four-way set-associative caches are simulated in place of direct-mapped caches. The change is expressed as the difference between invalidations for the set-associative cache minus the number of invalidations for the direct-mapped cache. The difference between the two cases is presented as a percentage of the direct-mapped base case.

The first notable feature of this table is the fact that the number of L1 invalidations does not increase when unshared L2 caches given higher associativity. This is independent of the the degree of sharing in the L1 cache. This is to be expected because sharing an L1 cache does not introduce any additional invalidations beyond the invalidations that come from other processors on the memory bus.

Increasing the amount of associativity in the L2 cache tends to decrease the number of invalidations. For the unshared L1 cache state, as the degree of L2 sharing is increased, we generally see a decrease in the number of invalidations. In fact, Barnes-Hutt and both water benchmarks show over a 98% decrease in the number of invalidations and a 99% decrease when L2 caches are shared at the 4:1 ratio. When the degree of L2 sharing is increase to 8:1, we see an increase in the number of invalidations. In fact both *fft* and *ocean-noncontig* show more invalidations in the set-associative case than in the direct-mapped case. A similar trend can be seen when sharing an L1 cache. As the

Benchmark	Change in Invalidations								
	1-1/1-1	1-1/2-1	1-1/4-1	1-1/8-1	2-1/1-1	2-1/2-1	2-1/4-1	4-1/1-1	4-1/1-2
barnes	0.00%	-98.87%	-99.21%	-87.83%	0.00%	-90.46%	-88.11%	0.00%	-81.74%
fft	0.00%	-54.62%	-39.71%	5.67%	0.00%	-26.26%	0.63%	0.00%	-21.59%
ocean-contig	0.00%	-68.84%	-83.83%	-40.16%	0.00%	-61.88%	-53.57%	0.00%	-57.01%
ocean-noncontig	0.00%	-53.79%	-56.06%	1.55%	0.00%	-27.05%	-7.77%	0.00%	-11.68%
water-nsquared	0.00%	-99.81%	-99.87%	-73.75%	0.00%	-90.90%	-76.18%	0.00%	-67.38%
water-spatial	0.00%	-99.45%	-99.66%	-76.66%	0.00%	-89.71%	-81.25%	0.00%	-80.41%

Table 2: **Change in L1 Invalidations** due to four-way set associativity

Benchmark	Change In Invalidations									
	1-1/1-1	1-1/2-1	1-1/4-1	1-1/8-1	2-1/1-1	2-1/2-1	2-1/4-1	4-1/1-1	4-1/1-2	8-1/1-1
barnes	-35.77%	-35.66%	-34.14%	-14.09%	-78.37%	-77.94%	-52.31%	-53.81%	-49.10%	-36.95%
fft	-3.28%	-2.50%	-2.58%	-2.27%	-9.77%	-9.78%	-5.66%	-8.14%	-7.33%	-6.89%
ocean contig	-6.43%	-6.50%	5.24%	-6.40%	-26.54%	-26.32%	-14.39%	-13.90%	-11.78%	1.44%
ocean noncontig	0.80%	0.66%	0.50%	-0.29%	-70.04%	-69.82%	-68.16%	-67.76%	-67.12%	-43.37%
water nsquared	-72.69%	-71.00%	11.67%	21.45%	-98.43%	-97.60%	-56.11%	-35.98%	-29.51%	-18.08%
water spatial	-82.89%	-82.89%	-82.03%	-15.40%	-95.53%	-95.42%	-46.83%	-49.30%	-42.31%	-31.61%

Table 3: **Change in L1 Invalidations** due to two-way set-associativity in the L1 cache.

degree of L2 sharing increases the change in L1 cache invalidations increases and then decreases. Table 3 shows the change in L1 invalidations when the L1 cache is made two-way set-associative. The trends here are similar to the ones discussed above (when L2 associativity was increased).

3.5 Constructive and destructive interference

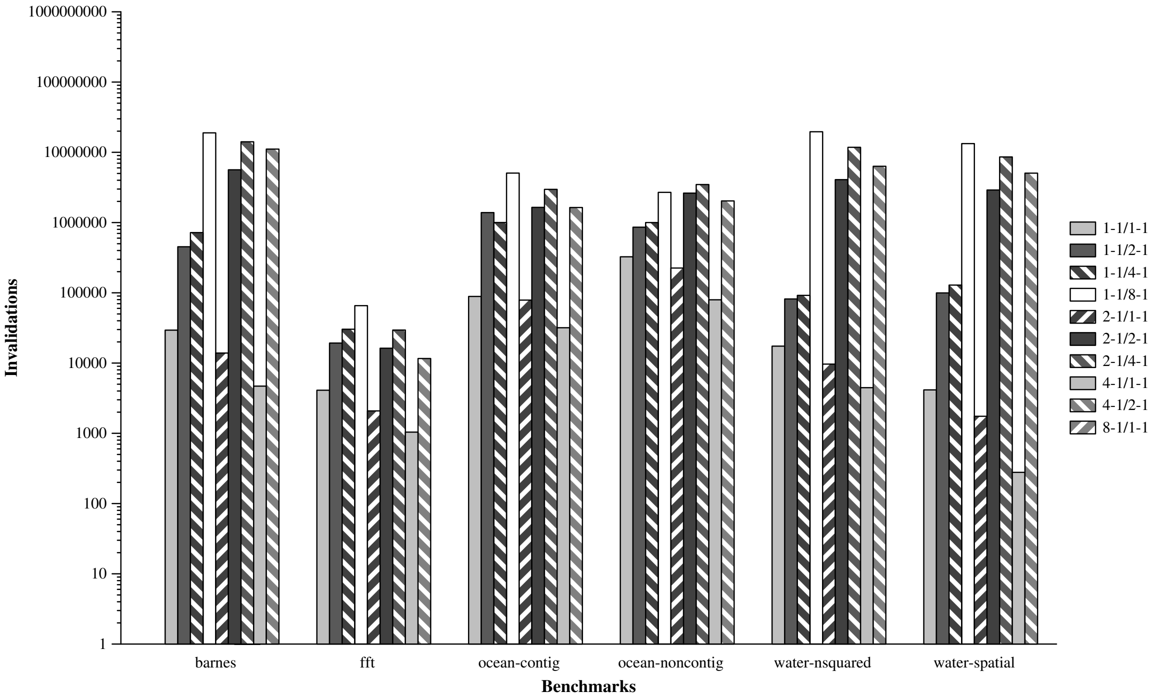

Figure 10: **L1 invalidations using direct mapped L1 and 4-way L2.**

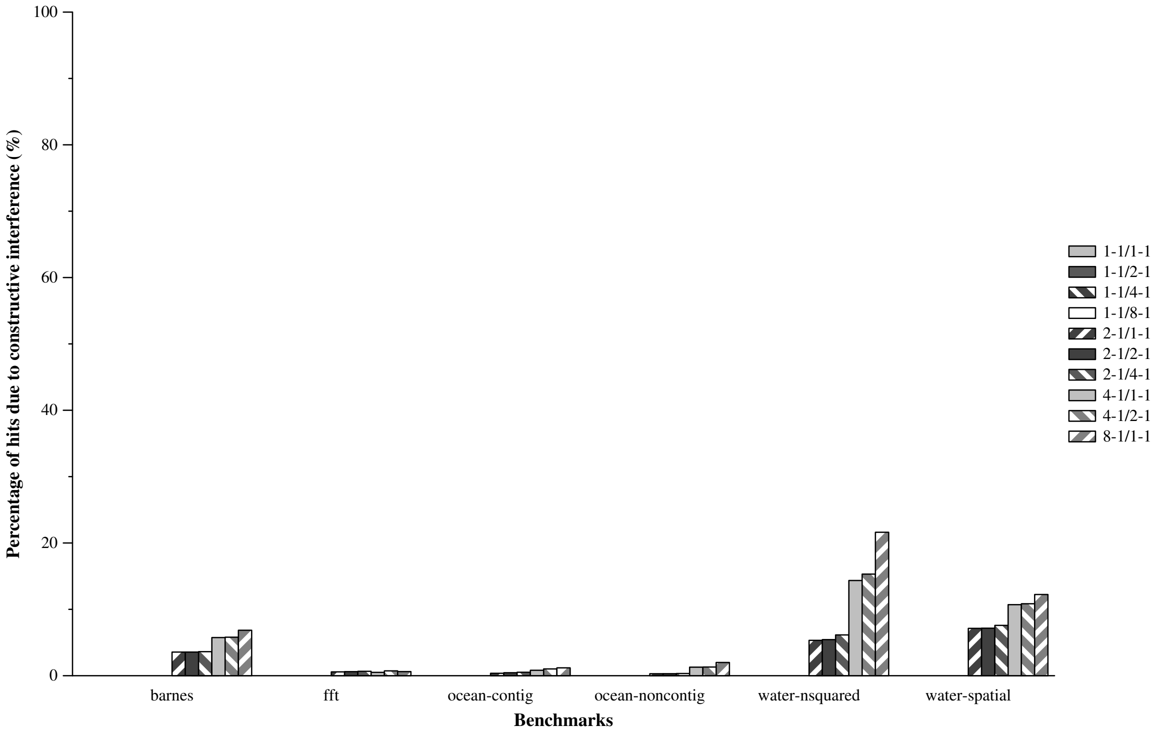

Figure 11: **L1 constructive interference using 2-way L1 and 4-way L2.**

3.5.1 L1 constructive and destructive interference

The L1 caches show only a small amount of constructive and destructive interference. Intuitively, the low interference effect is due to the small size of the L1 cache relative to the working set size. Figure 11 shows two interesting sets of sharing configurations for constructive interference. First, when the L2 is not shared, the constructive intereference as a percentage of hits increases. However, the increase is small in absolute terms. Sharing of L1 caches increases the opportunity for constructive interference. Second, the L1 cache is insensitive to changes in L2 cache sharing.

For destructive interference, when L2 caches are shared, Figure 12 shows destructive interference decreases as a percentage of total misses. We hypothosize this is due to invalidations from the shared L2 caches because of the effect of maintaining inclusion, which was explained in section 3.4.

3.5.2 L2 constructive and destructive interference

The L2 caches exhibit different results due to the larger size of the L2 caches and to the fact that L2 caches are not invalidated due to inclusion, unlike L1 caches. Unlike Figure 11, Figure 13 shows a dramatically higher percentage of

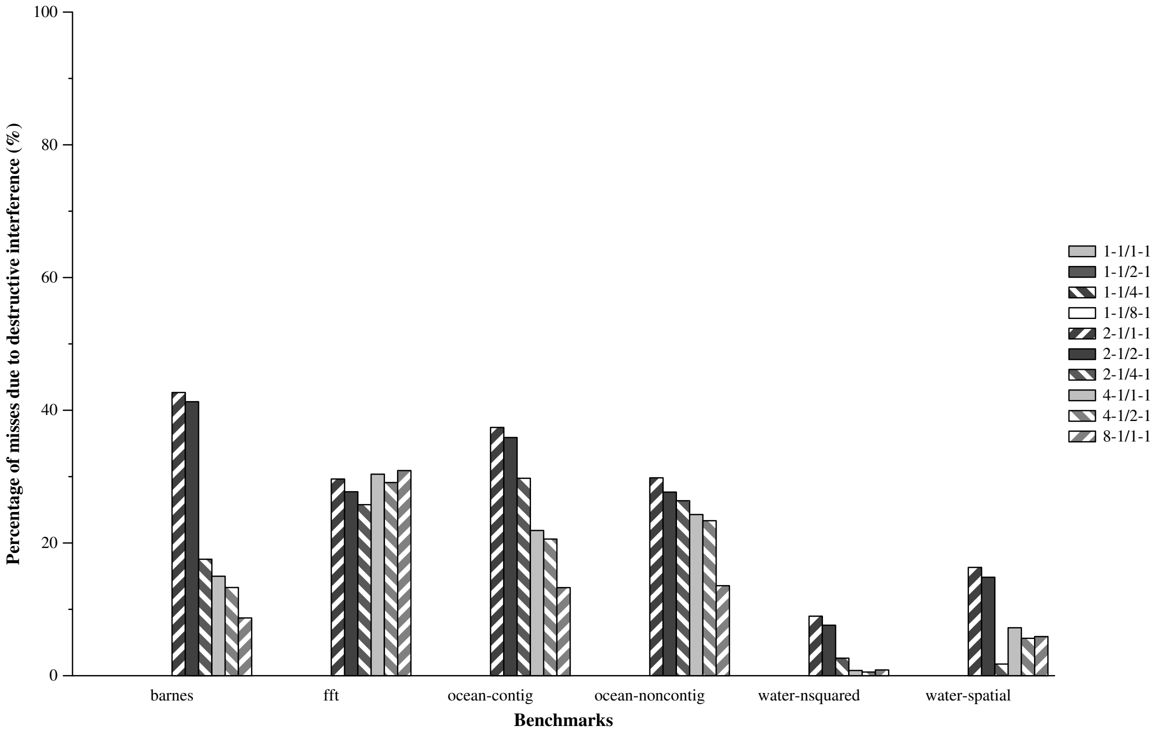

Figure 12: **L1 destructive interference using 2-way L1 and 4-way L2.**

hits due to constructive interference. The large size of the L2 accomodates the working set of the sharing L1 caches. The combined working set has good temporal locality between each L1 cache's working set. Thus, the processors are essentially prefetching data for each other in an uncoordinated fashion.

The L2 caches, as shown in Figure 14, exhibit a higher cache destructive interference as a percentage of the misses. Destructive interference is a form of conflict misses. Conflicts misses are a substantial contributor to overall misses as seen in Figure 8.

4 Conclusion

This paper demonstrates the effects of cache sharing on cache miss rates, inter-cache invalidations, and constructive and destructive interference patterns. Our results indicate that the best cache hierarchy configuration for a CMP would most likely be the following:

- No sharing of L1 caches between processors.
- Share the L2 cache between processors.
- Do not enforce inclusion between the L1 and L2 caches.

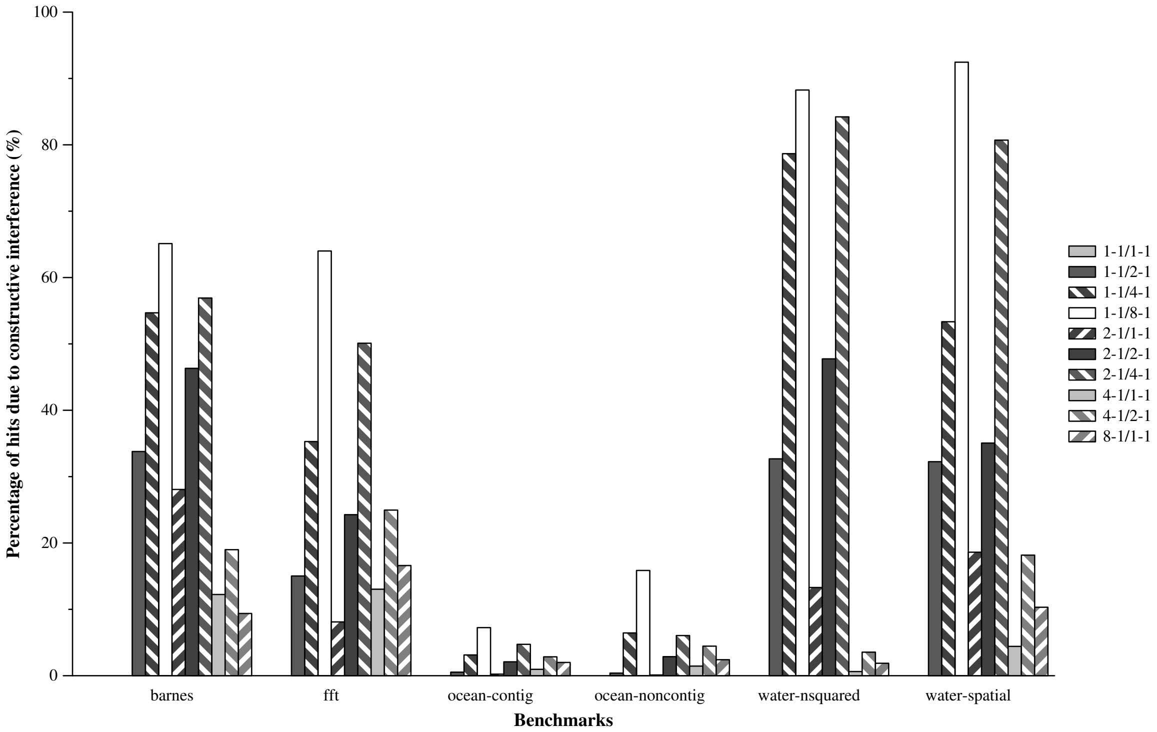

Figure 13: **L2 constructive interference using 2-way L1 and 4-way L2.**

- Multiple sets for the L1 and L2 caches.

The reason for not sharing the L1 is evident in the miss ratio graphs and the destructive interference table. Since applications tend to be written to exploit the L1 cache, good performance of these applications is dependent upon good L1 hit rates. Also, there relatively small size leads to excessive misses due to conflicts if L1 cache sharing takes place. L2 caches, on the other hand can be much larger and handle much more sharing. The L2 should also be accessed much less than the L1 so a reduction in its performance should not be nearly as detrimental to overall performance.

The reason for not enforcing inclusion should be obvious from the invalidation graphs. Our simulator enforces inclusion so if multiple L1 caches mapped to an L2, the number of invalidations for those L1 caches rises drastically. The reason for this is because lines being replaced in an L2 must be invalidated in all higher L1 caches. This replacement can be alleviated by having more associativity in the L2 cache but it is still a major source of invalidations in the L1 cache.

The major motivation for doing sharing at the L2 level of the cache is that it allows much faster inter-process communication than sharing at the memory level. Also, constructive interference can actually be quite beneficial in a

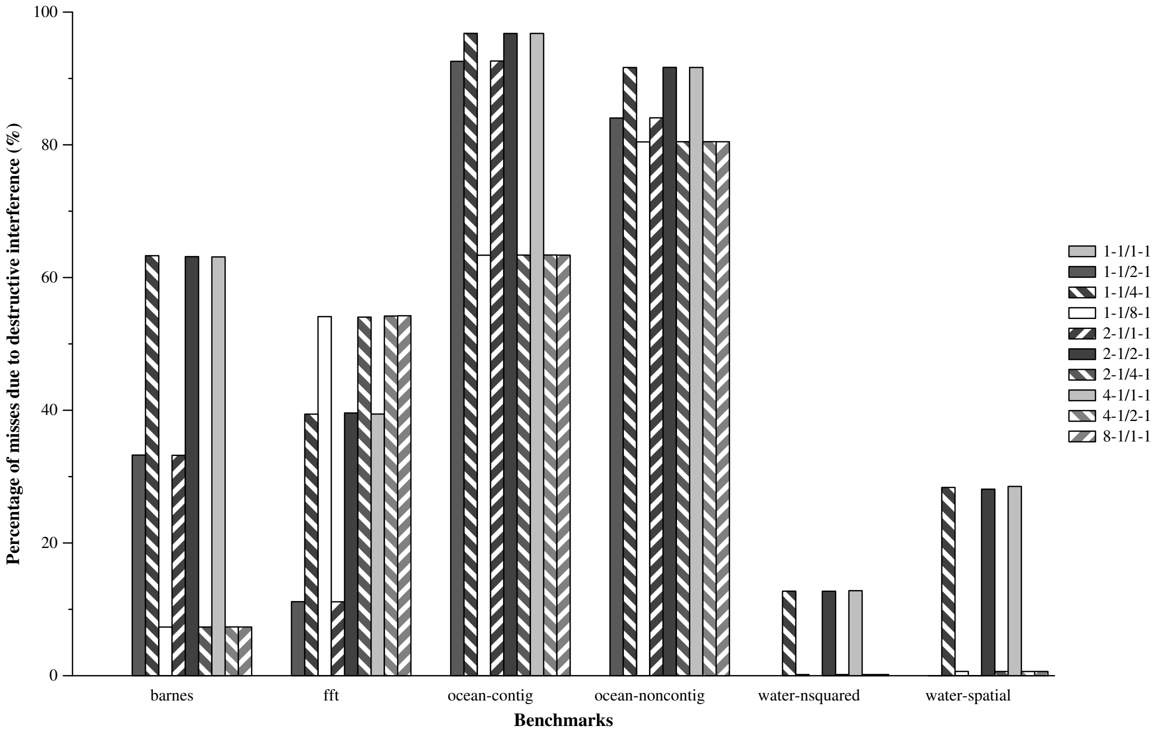

Figure 14: **L2 destructive interference using 2-way L1 and 4-way L2.**

large L2 cache because a significant amount of data can be stored by one processor and used by another.

References

[1] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 3–14, June 1998.

[2] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[3] Douglas Berger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report 358, University of Wisconsin–Madison, June 1997.

[4] Keith Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, 30, October 1999.

[5] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39–50, June 1998.

[6] Naraig Manjikian. Multiprocessor enhancements of the simplescalar tool set. *Computer Architecture News*, 29:8–15, 2001.

[7] Basen A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd Annual Internation Symposium on Computer Architecture*, pages 67–77, May 1996.

[8] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.