ECE1718 Project Final Report
Improving Data Locality During Thread-Level Speculation

Adrian Tam and David Tam

May 7, 2003

Abstract

Locality conflict is a major problem during thread-level speculation (TLS). This paper addresses three potential techniques for reducing data cache misses, namely universal prefetching, ORB prefetching and prefetching on speculative violation. Universal prefetching works by prefetching clean cache lines from the unified cache to all data caches when one of the data caches suffer a speculative read miss. ORB prefetching works by prefetching modified data at the end of a processor's speculative execution. Prefetch on speculation violation will prefetch the invalidated data when the epoch recovers from a violation. The experiments show that universal prefetching is generally efficient in reducing both locality misses and cycle counts. ORB prefetching reduces the number of locality misses, but it increases speculation commit times Prefetching on speculative violation does not have a big impact on the system even in the most optimistic model. The lesson learned from this project is that it is pertinent to consider all factors, including communication cost and latency cost, when designing cache optimization techniques.

1 Introduction

Over the past decade, computer architects have presented many interesting ideas to improve system performance. These ideas include simultaneous-multithreaded processors (SMT) [7] and chip multiprocessors (CMP) [2]. In fact, modern processors, such as the Intel Xeon Processor [3] and Alpha 21464, already implement these ideas in their CPU design. The end-users, however, have yet to see the full potential of these innovative ideas. The problem lies in the fact that programmers and compilers have difficulties in parallelizing applications. Compilers often cannot prove whether threads are independent in irregular numeric and non-numeric applications.

Thread-level speculation (TLS) [6] fulfills an important need in the realm of compiler optimization. It allows for the automatic parallelization of general-purpose (non-scientific) applications, decomposing a single thread of execution into multiple threads and allowing each one to execute speculatively. Steffan et. al. show that using TLS techniques can provide speedups of 8% to 46% [6] for a four-processor single-chip multiprocessor.

1.1 Locality Conflict

TLS involves artificially dividing a single thread of execution into multiple threads. Consequently, there is a high probability that data is shared among these multiple threads. The architecture, however, does not take advantage of the spacial and temporal localities between the multiple threads. In this document, we refer to data cache miss during speculative mode as locality miss. There are many cases in that, during speculation, one processor suffers L1 (data) cache miss while the cache line is already in another processor's private L1 cache, which we will refer to as locality conflict. An illustration of locality conflict can be found

in **Figure 1**. Processor A suffers a cache miss for cache line 0x1234. However, processor D's L1 private cache already has a copy of the data. Ideally, processor A should already have a copy of this cache line so that it does not suffer such L1 cache misses.

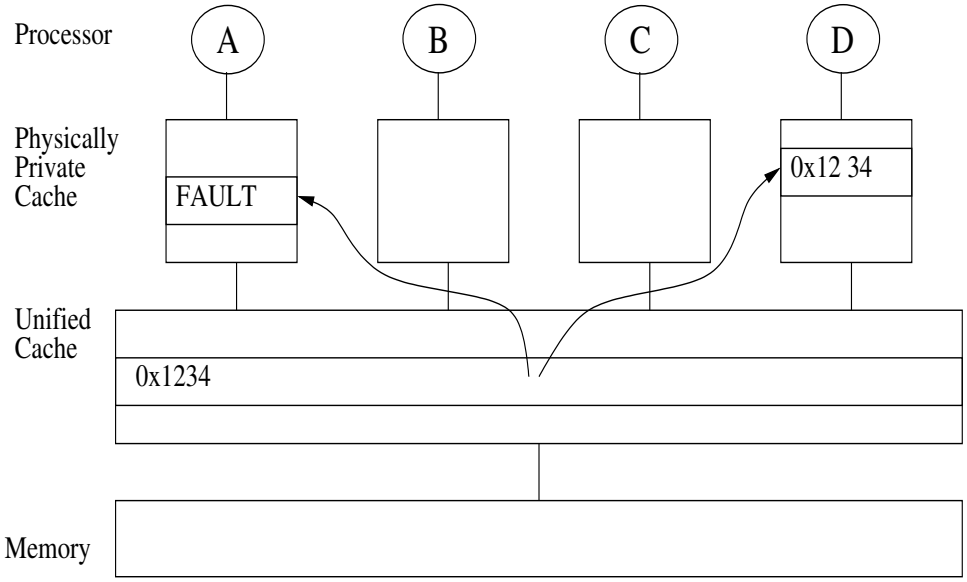

Figure 1: Illustration of locality conflict.

A brief examination of SPEC CPU95 and SPEC CPU2000 benchmarks, as summarized in **Figure 2**, shows that locality conflicts happen frequently. In fact, locality conflicts comprise of 16% to 96% of all the locality misses. Hence, there is a great incentive in reducing locality conflicts during thread-level speculation. We believe that reducing the locality conflicts can improve performance during speculative execution.

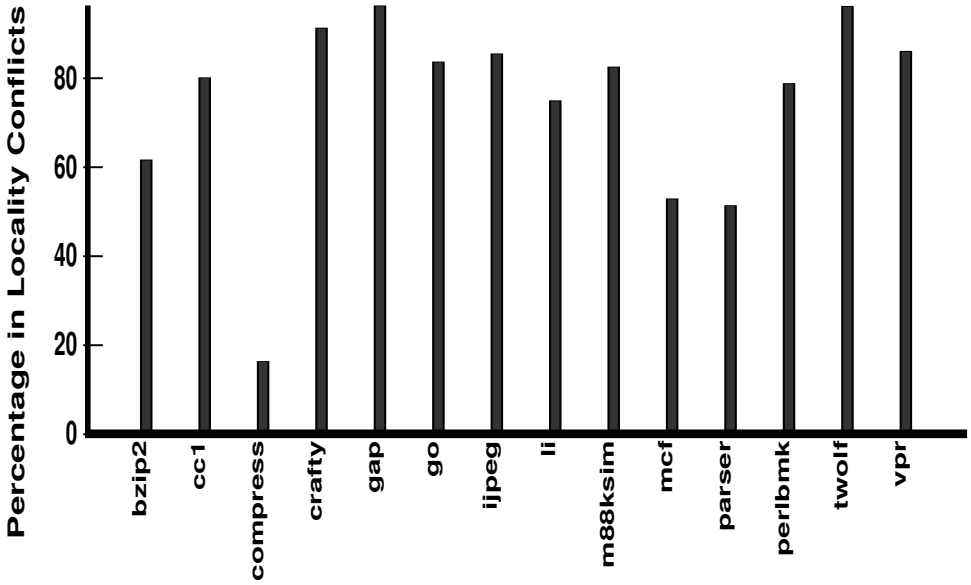

Figure 2: Percentage of locality conflicts from all locality misses.

1.2 Related Work

Many researches have focused on improving TLS techniques. Steffan et. al. [6] developed TLS techniques which delay speculative loading until corresponding speculative stores are completed. When speculative stores are completed, the logically-next epoch is signaled to consume the value. Also, multiple writers are supported by merging writes during data commit.

Prvulovic et. al. [4] tries to remove the bottleneck on thread-level speculative system for scalable machines. Three areas of bottlenecks are identified, namely in task commit, speculative buffer overflow, and speculation-induced traffic. First, Prvulovic provides low complexity task commits that completes in constant time. The speculative buffer overflow problem is addressed by having an unlimited-size overflow area in local NUMA network that stores uncommitted tasks. The problem of speculation-induced traffic is solved by having a No-Traffic cache state. This allows data to be addressed locally for certain data access patterns.

Roth and Sohi [5] improves TLS performance by selecting the optimal thread to pre-execute. Parameters for choosing the optimal thread include overhead, latency tolerance as well as completion time. Thus, our project can potentially complement their techniques by allowing more aggressive thread speculation.

Figueiredo and Fortes [1] discuss the merit of speculative distributed shared-memory (DDSM) multiprocessors. DDSM is designed to support thread-level speculation. This is accomplished by extending the existing L2 data cache for speculative protocols. In addition, DDSM, with an extra *sharers* vector to distinguish readers from writers, does not have to send additional messages to the sharers when checking for dependencies. Instead, it directly uses information from the directory structure.

1.3 Goals

The goal of this project is to reduce data cache misses during thread-level speculation. We propose three solutions to reduce speculative conflicts. The feasibility of these solutions are examined through simulation and prototyping. Due to time constraints, we have only focused and tested the techniques for a shared memory chip-multiprocessor with an invalidation-based cache coherence scheme. We believe, however, that these techniques may be extended to multi-chip multiprocessor.

The remainder of this paper is as follows. Section 2 describes the motivation and design of our three prefetching techniques. Section 3 provides the experimental setup. The experimental results are presented and discussed in Section 4. Section 5 summarizes the lessons learned and presents the conclusions.

Since our discussion focuses on extending the ideas of Thread-Level Speculation and uses many of its terminologies, we encourage readers to refer to [6] for more information on TLS.

2 Design

The three techniques that we examine for this project are universal prefetching, ORB prefetching and prefetching on speculative violation. In this section, we will describe the above techniques as well as analyze their merits.

2.1 Universal Prefetching

2.1.1 Motivation

Imagine a sequential program that reads some data from the main memory. The program will suffer 1 cold miss per cache line for each level of cache. If we extend this scenario to TLS, intuition tells us that the program should not suffer more cache misses than before. This is not necessarily the case in the current design. Let say that we have a 4 CPU CMP, and each CPU reads some common data. Each CPU will suffer 1 L1 cold cache miss because the CPUs' L1 caches are private. Hence, for a 4 CPU CMP, there will be 4 L1 cache misses instead of 1 L1 cache miss. The increase in L1 cache misses reduces the potential benefit of running the program in speculation.

The universal prefetching mechanism attempts to alleviate this bottleneck. Its premise is that, based on temporal locality, data accessed in one epoch is more likely to be accessed in the next epoch. Universal prefetching works as follows: When one CPU suffers a cache miss, the L1 cache will try to fetch the information from the L2 cache. Instead of sending the data only to the requesting L1 cache (which is illustrated in **Figure 3**), the L2 cache will send the cache line to all L1 caches. The new method is illustrated in **Figure 4**.

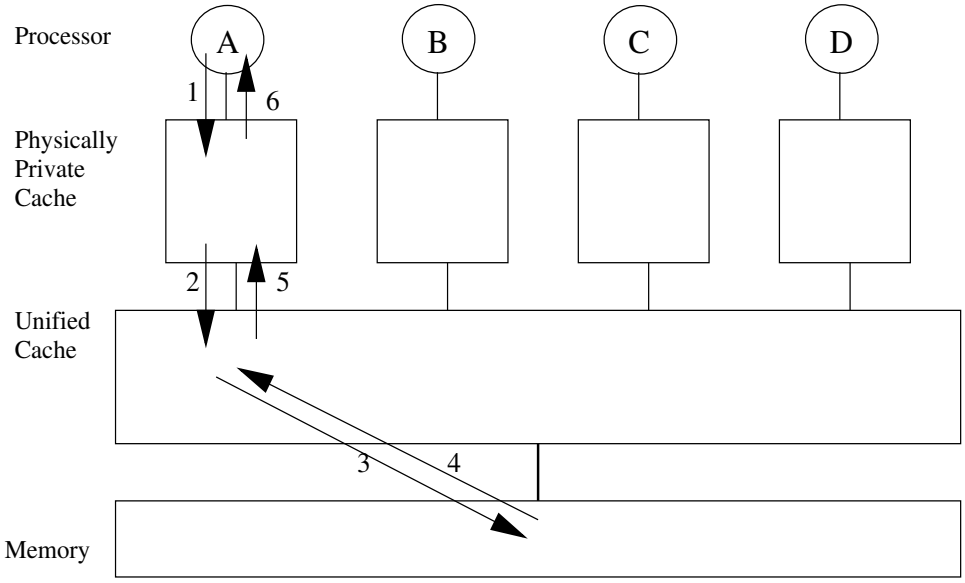

Figure 3: Original miss handler.

2.1.2 State Coherence

The next question that needs to be addressed is - when should universal prefetching be performed? If the data is going to be modified in the near future, then universal prefetching does not improve performance. Not only the other CPUs cannot use the data (as the data becomes stale quickly), it also increases write latency since the original CPU has to send invalidation messages to all other CPUs. Therefore, universal prefetching is only applicable to speculative read, not speculative write.

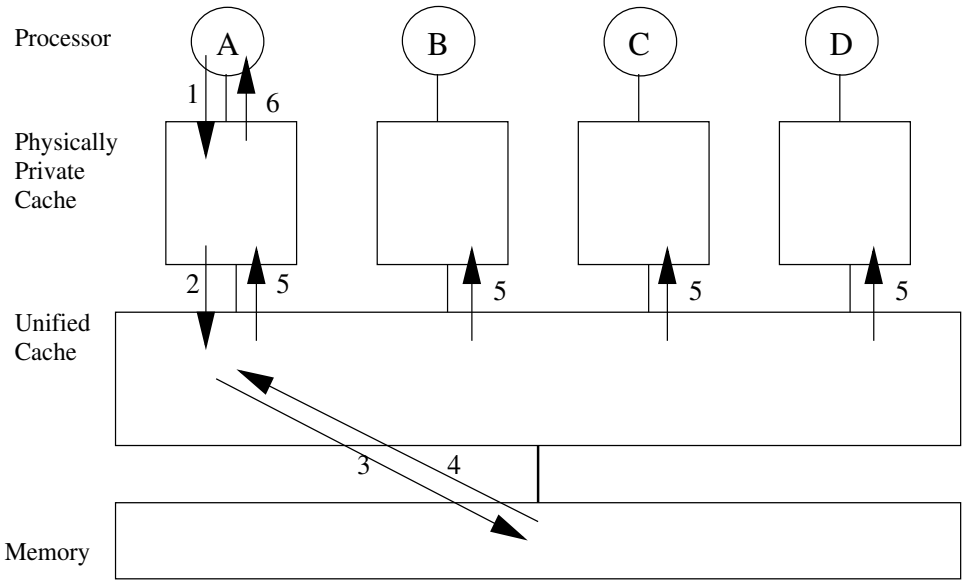

Figure 4: Universal prefetch miss handler.

Another aspect that needs to be considered is - how does the system know whether the data will soon be modified (which means the cache line should not be universally prefetched). The answer lies in the history of the cache line. If the unified cache is dirty, then it implies the data has been modified in the past - therefore, will likely be modified again in the future.

After considering the above questions, we have implemented mechanisms, based on the cache's state, that determine whether to invoke universal prefetching. They are illustrated in **Figure 5** and **Table 1**. During a speculative read miss, the L1 data cache that suffers the miss will send a *prefetch* request to the underlying L2 unified cache. Based on the unified cache's state for this particular cache line, there are two choices. If the unified cache line is dirty or modified, then the unified cache proceeds as a normal read - which notifies the L1 owner that the line will be read by another CPU. After the message is sent and the appropriate coherence protocol is observed, the data will be delivered to the requesting L1 cache. Otherwise, if the unified cache is in invalid, shared or exclusive state, then the unified cache will send a message to all CPUs (that is not listed in the cache directory) that the receiver should prefetch the specified cache line. Upon receiving this message, the receivers, after maintaining the coherence protocol, prefetch by issuing a *read* request.

Message	Description
Send_to_lower:Fetch	Send message to fetch cache line from memory
Send_to_all:Prefetch	Send messages to all L1 caches for prefetching the cache line
Send_to_owner:NotifyShare	Send message to the cache line's owner that the cache line is now in shared state
Send_to_owner:Read	Send message to the cache line's owner that the cache line is read by another processor

Table 1: Explanation of unified cache's actions after receiving prefetch request.

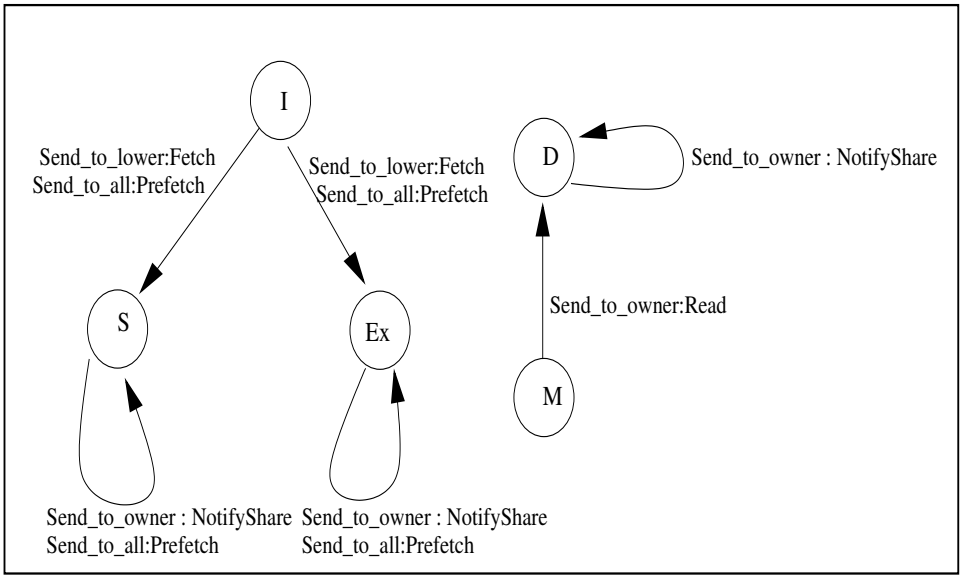

Figure 5: Unified cache's response to prefetch request.

2.1.3 Impact on the System

There are both advantages and drawbacks in using universal prefetching. The number of cold cache miss can potentially be reduced by a factor of N, where N is the number of L1 caches in each unified cache. Instead of having all L1 cache suffering its own individual cold L1 cache miss, there will be only 1 L1 cache miss. On the other hand, the L1 cache miss latency may be increased. The underlying communication network in the unified cache may need to communicate with all L1 caches. Another potential problem for universal prefetching is that it may force useful data out of the data cache. This problem may be especially severe when the data cache is small. However, modern processors have large L1 caches. It is our experience, as will be evident in the evaluation section, that 32KB of data cache is sufficient. One may argue that the hardware cost will increase - as we have introduced a new type of communication message. This is not significant because the associated cost is minimal. It introduces only one additional request type, which corresponds to a maximum of 1 bit per message.

2.2 ORB Prefetching

2.2.1 Motivation

Although the universal prefetch technique is intended to target locality conflicts, it does not target the entire set of potential prefetch candidates shown in **Figure 1**. From this set, a subset is in the dirty state, signifying that the data line was brought into some other processor's data cache and subsequently modified. **Figure 6** illustrates the proportion of the dirty subset for each benchmark. This proportion cannot be handled by the universal prefetch technique.

Although universal prefetching will be triggered when a processor reads a targeted cache line, it is not triggered when the processor subsequently modifies that particular cache line. The ORB prefetching technique described in this section specifically

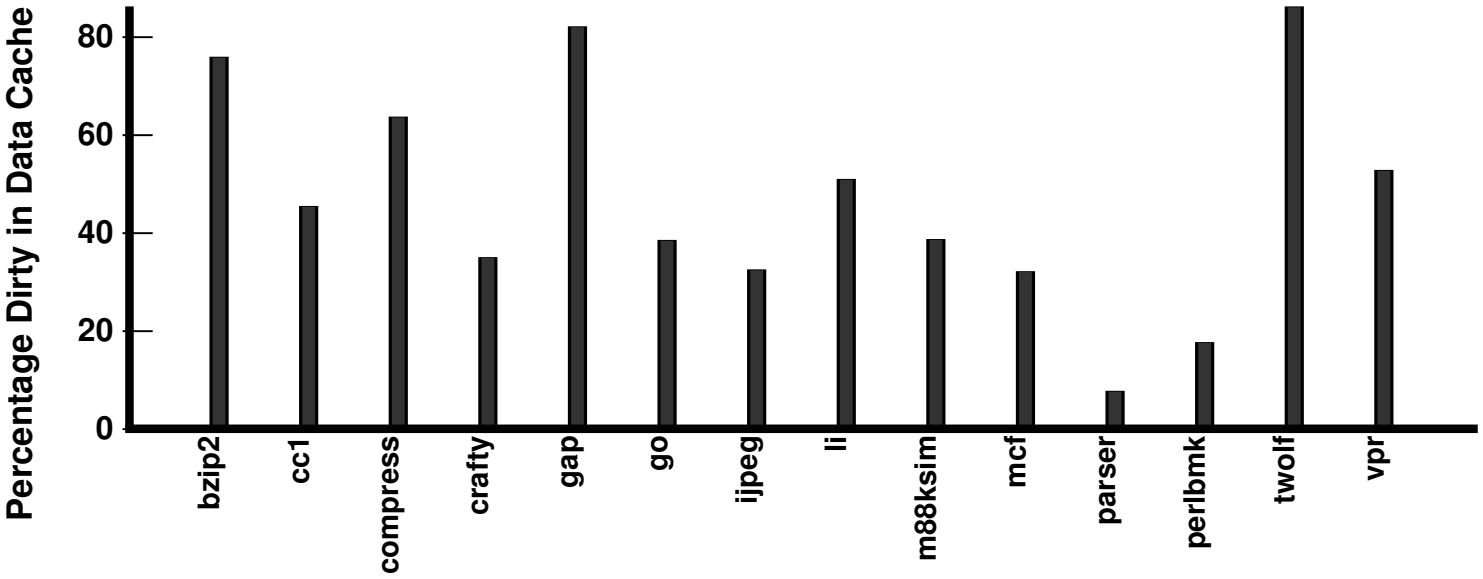

Figure 6: Percentage of locality conflicts that are in the dirty state.

targets this dirty subset. This technique is complimentary and mostly non-overlapping with the universal prefetch technique since it targets a non-overlapping set of candidates. It is called the ORB prefetching technique since it makes use of the *ownership-required buffer* data structure in TLS Cello.

To provide a concrete example of motivating statistics, let us examine the cache miss characteristics of *gap*. From **Figure 2**, we see that when *gap* suffers a data cache miss, 85% of the time, the data cache line is present in another processor's data cache. From **Figure 6**, we see that of this 85% proportion, 80% of this is in the dirty state. Therefore, there is significant potential for cache miss reduction by using the ORB prefetching technique. In contrast, *parser* shows very little potential. As shown in **Figure 2**, when *parser* suffers a data cache miss, only 55% of the time is the cache line present in another processor's data cache. From **Figure 6**, we see that only 10% of that 55% is in the dirty state.

2.2.2 Design

An important criteria that must be considered in data writes is the frequency of the operation since each time the write operation is invoked, the data and the cache is actively modified. In contrast, the operation frequency criteria is perhaps less important in data read operations since the operation does not actively modify the cache. The cache could be viewed as a passive entity in this case.

As well, during speculative operation, writes to a processor's data cache are not completely visible to other data caches. For processors that are in the speculative execution mode and have a larger epoch number, these write are visible to ensure that data dependencies are maintained. For processors that are in the speculative execution mode and have a smaller epoch number, these writes are not visible. For processors that are not in speculation mode, these writes are also not visible.

Factoring write frequency into the design, we trigger the ORB prefetching technique at the end of a processor's speculative

execution mode. At that point in time, the processor's changes are made visible to all processors and caches. As well, we predict that the processor is very likely done modifying its data cache. In other words, the frequency of write operations to a particular cache line has transitioned to a low value and will remain low for a relatively long period of time. This is an optimal point in time in which to send the updated date to other data caches.

In other words, targeting dirty cache lines with low write frequency is our approach. In contrast, targeting dirty cache lines with high write frequency causes a lot of wasted cache traffic since the previous version of the data is invalid and was transferred without being used.

We believe that at the end of speculative execution, the new and visible data produced by the processor will most likely be needed by another processor and its associated data cache. This is mostly an intuitive reason since, at a high level of abstraction, processors are given data to process, produce some kind of output, and that output is eventually consumed by another entity at a later time.

For simplicity in terms of hardware realization, the targeted data is sent to all data caches. Designing and implementing more intelligent mechanisms would increase complexity and latency. For example, data cache A could keep track of which other data caches actually consume the data produced by A. Then, data cache A could send updates to only those data caches that exceed some threshold value related to use of data from cache A. Maintaining such a data structure in hardware may be expensive, complex, increase latency, and decrease performance. The ability to watch all other data caches and their actions with their associated processors may be an unrealistic feature to implement.

2.2.3 Implementation

We attempt to reuse as much of the existing hardware mechanism as possible. This would allow us to leverage existing real-estate space on the chip. As well, the implementation would be less intrusive and more acceptable to the VLSI hardware chip designers.

The *ownership-required buffer* (ORB) and the *notify-modified-required buffer* (NMRB) are two existing data structures used by the TLS mechanism to ensure cache coherence. These two structures are reused by the ORB prefetch technique to determine which data caches lines have been modified and need to be propagated to the other data caches. The ORB specifies cache lines that have been modified speculatively and also exists in some other data cache. Ownership of the data cache line is desired by the current data cache so as to maintain cache coherence. The NMRB specifies cache lines that (1) have been modified by the data cache, and (2) that no other copies of the cache line exist in other data caches or the unified cache. The unified cache must be notified that our data cache has modified the cache line and has the most recent copy. For cache coherence purposes, this prevents the unified cache from serving its own out-of-date copy to other requesting data caches. Rather, the unified cache will first obtain the latest copy from the appropriate data cache. In combination, the ORB and NMRB specify the majority of the cache lines that (1) have been modified during speculative execution, and (2) that should be made visible to other caches.

We realize there may be cases where a cache line has been modified during speculative execution and a corresponding entry

does not appear in the ORB or NMRB. For example, consider a cache line in data cache A that is marked as modified and exclusive in the unified cache before it enters speculative execution mode. During speculative execution, the data cache line is modified. At the end of speculative execution, a corresponding entry does not need to appear in the NMRB and perhaps not the ORB either. A full solution would require a full linear, sequential scan of all lines of the data cache, defeating the performance optimization purpose of the ORB. We feel that the coverage and performance provided by the ORB and NMRB outweighs the cost of the full solution.

In the existing TLS mechanisms in Cello, upon ORB or NMRB flushing, each entry is placed into the data reference buffer/queue to allow for asynchronous request handling, reordering, and optimization. Modifications were made so that when the data reference handler dequeues a request and it turns out to be an ORB request, it triggers the prefetching mechanism. Note that due to the asynchronous buffer handler, this trigger point should not affect ORB flushing time, which could delay home-free token passing.

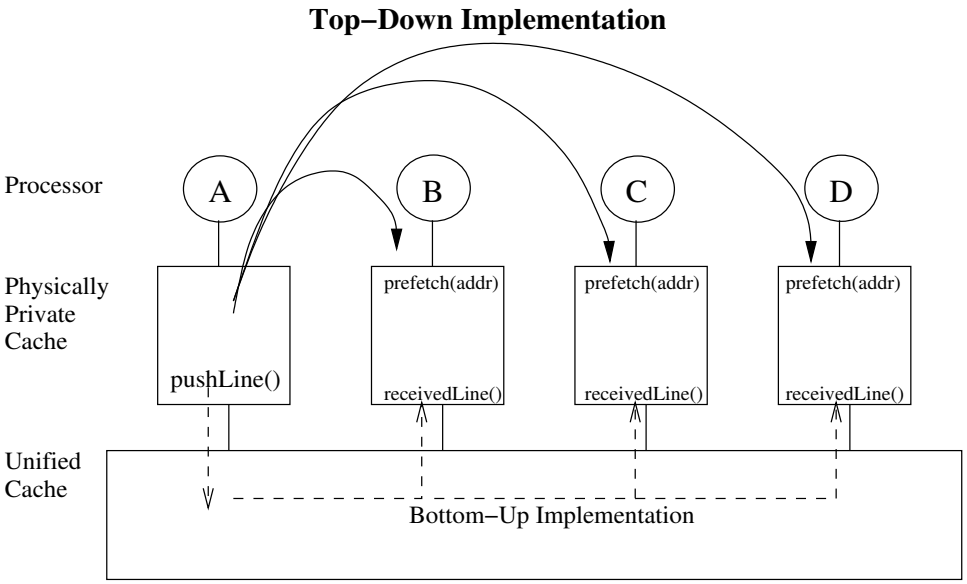

Figure 7: Top-down vs bottom-up implementation of ORB prefetching technique.

The actual trigger mechanism reuses the same interface as that used by the processor to send requests to the data cache. This implementation may be considered as a top-down trigger approach. A high-level view of the approach is shown in **Figure 7**. Processor A is depicted as containing modified data that is prefetched into other data caches. The advantage of this approach is that it reuses many of the existing communication and coherence mechanisms between the data cache and unified cache. Only a minimal amount of change is required to the top-end of the data cache interface. If a prefetch interface in the data cache has already been made available to the processor, perhaps it could be simply reused with any modifications. In contrast, the bottom-up approach would require adding additional state, and logic to handle requests send from the unified cache up to the data cache.

As a note of caution, the implementation of the prefetch function of the data cache make use of a regular read request to the unified cache rather than a speculative read request. Regular reads are requests independent of whether the processor is speculative or normal operational mode. In contrast, by prefetching a cache line into speculatively loaded mode, the processor may suffer *false violations*. That is, the current processor would be enforcing dependencies of the speculatively load cache line even when the line is not used by the processor. Only when the processor actually accesses that particular cache line for read purposes will that cache line transition to speculatively loaded mode.

2.2.4 Trade-Offs

A potential major disadvantage of this simple technique is that there is an increase of bus traffic, compounded by the possibility that the majority of the increase is of useless traffic. Since data is sent to all data caches, the corresponding processors may not make use of the updated data. However, this traffic could be set to low priority and disabled when cache traffic volumes are above some threshold. In terms of the top down approach, prefetch requests can also be placed in a low priority data reference queue serviced less frequently than the normal queue.

Another disadvantage of this simple technique is that it may increase the latency of write operations. In particular, if a cache line is present and not dirty in all data caches, and processor A wishes to write to that particular cache line, then invalidation and acquiring exclusive ownership of the cache line may take longer than if it was present (and not dirty) in one other data cache. We hope that the access pattern of a cache line is mostly "write, read, read, read, read, read, etc...". Under such a pattern, a speculative processor may produce a data value and write it to a cache line. When the speculation mode is over and the data is prefetched to the other data caches, the other processors should mainly be reading this value. Perhaps an intelligent compiler can optimize for these scenarios, much like compilers can optimize performance by cache line padding and intelligent data placement.

Another concern is potential cache conflicts, in that a useful line is evicted from a data cache and replaced with a useless line that was prefetched using the ORB prefetch mechanism. Data caches with some level of set-associativity can help reduce this potential problem. As will be shown in the results, cache conflicts did not occur.

2.3 Prefetching on Speculative Violation

2.3.1 Motivation

A processor executing in speculative mode suffers speculative violation when a dependency violation is detected. At this point, the processor is halted, all speculatively modified cache lines are invalidated and simply discarded. Speculatively loaded and unmodified cache lines are simply converted to "regularly loaded". That speculative execution, called an epoch, is restarted after perhaps waiting until a certain condition is fulfilled.

The restarted epoch will mostly traverse the same execution path and access the same cache lines. For the caches lines that were set to "regularly loaded", cache hits will occur. On the other hand, for cache lines that were set to invalidate and simply

discarded, cache misses will occur.

Rather than simply discarding (and invalidating) a speculatively modified cache line at violation time, there is an opportunity to pre/re-fetch these cache lines and prevent future cache misses.

2.3.2 Design and Implementation

The design of this technique is straight-forward. The trigger point of this technique is at speculative violation time. When a speculatively modified cache line is set to invalid, a data reference request is constructed and added to the data reference handler's queue. Again, since handler operates asynchronously, this trigger point should not affect violation time consumption. Finally, the prefetch function of the data cache issues a regular read request rather than a speculative read request.

Due to time constraints, accurate modeling of this prefetch technique was not implemented. Rather, only initial a quick implementation that effectively implemented 0-latency ideal prefetching was developed. This 0-latency implement is still useful in that it can indicate the upper bound in performance improvements. A top-down implementation that should be relatively easy to develop since it can re-use some of the infrastructure developed in the ORB prefetch technique.

2.3.3 Trade-Offs

A disadvantage of this prefetch technique is that a re-executed epoch may follow a different execution path. Therefore, it may not encounter the same cache lines again.

In this technique and in any prefetch technique, the problem of increased cache line invalidation latencies is present. As well, the problem of cache line conflicts is also present.

3 Experimental Setup

The merits of the prefetching methods are evaluated through detailed simulation. The simulator, Cello, models a 4-way issue, out-of-order, superscalar, 4 CPU chip multiprocessor. Details on individual parameters can be found in **Table 2** and **Table 3**. The benchmarks are from the SPEC CPU95, SPEC CPU2000 (both integer and floating point) test suite. The benchmarks' binaries were generated from a compiler designed specifically for Cello. When we refer to cycle counts, we only consider the cycle counts in the speculative region. This is sufficient because all of our prefetching techniques are designed specifically for speculative regions.

4 Experiment

The experiment section presents three important system characteristics, namely reduction in dirty conflicts, reduction in locality misses and improvement in cycle counts. The first characteristic is mostly relevant to ORB prefetching while the latter two can provide insights into all three prefetching techniques.

Number of CPU	4
Functional Units	2 INT, 1 BRANCH, 2 FP, 1 LDST
Reorder Buffer Size	128
Branch Prediction	Gshare(8 bit history, 16K entries)
Integer Multiply Latency	12 cycles
Integer Divide Latency	76 cycles
Floating Point Add and Multiply	2 cycles
Floating Point Divide	15 cycles
Floating Point Square Root	20 cycles

Table 2: Experimental system parameters.

Number of Data Caches	4
Number of Unified Cache	1
Data Cache	32 Kbytes, 2 ways associative
Cache Replacement Policy	LRU
Data Cache Miss Latency and Fill Time	0 cycles and 4 cycles
Unified Cache Size	2048 Kbytes, 4-way set-associative
Unified Cache Cache Miss Latency and Fill Time	40 cycles and 4 cycles
Interconnection between unified and data cache	crossbar

Table 3: Experimental Memory parameters

4.1 Locality Conflicts

The first set of results for the ORB prefetching technique is shown in **Figure 8**. These results provide answers to the question raised in **Figure 6**, where typically a significant percentage of conflict misses were in the dirty state. **Figure 8** shows the reduction in the number of dirty misses. For example, the number of dirty conflicts in *bzip2* was reduced by 80%. In general, the results show that the ORB prefetching technique achieves its goal, in that it is effective in reducing the number of dirty conflicts.

4.2 Locality Misses

The results, as shown in **Figure 9** and **Figure 10** indicate that the three techniques are generally effective in reducing the number of locality misses during speculative execution.

In these graphs, *upf* refers to the universal prefetch technique, *orb* refers to the ORB prefetch technique, *combo* refers to the combination of the universal prefetch and ORB prefetch techniques, *viol* refers to the prefetch on violation technique, and *orb0* refers to zero-latency ORB prefetching.

Universal prefetching is shown to be more effective than ORB prefetching in most cases. The combined technique (universal and ORB) offers additional improvements. The prefetch on violation technique is highly effective only on a few workloads.

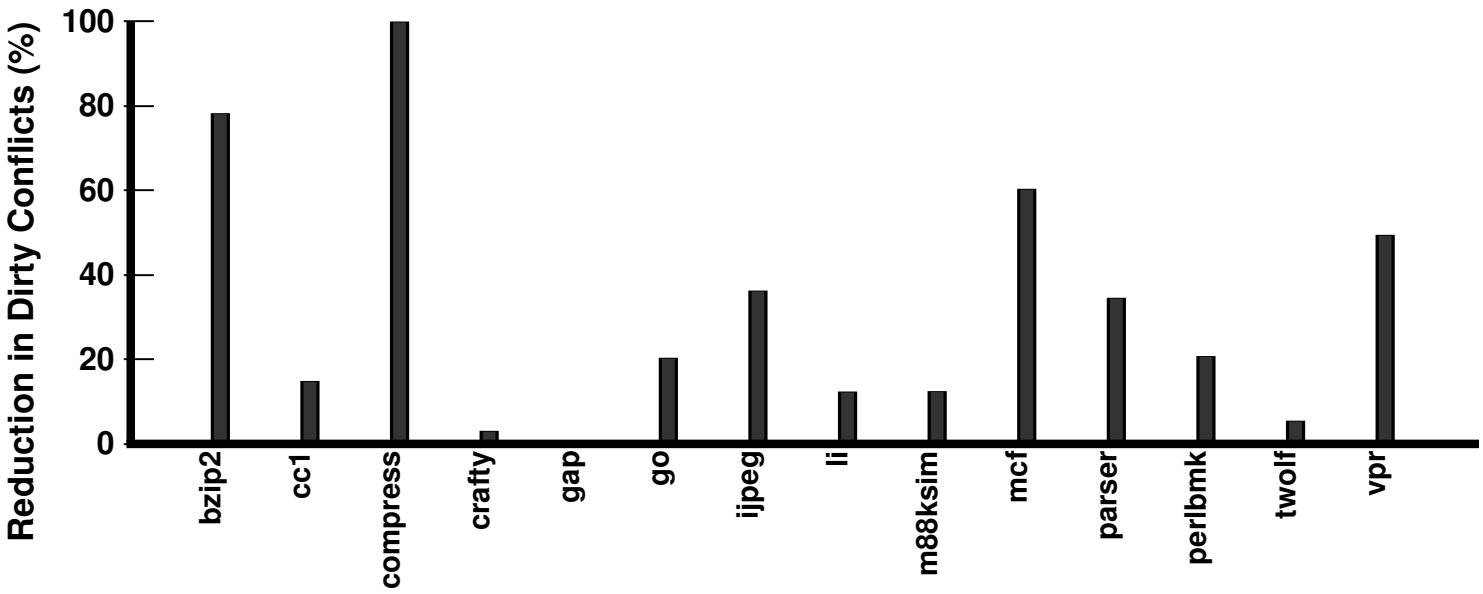

Figure 8: Reduction in dirty conflicts.

4.3 Cycle Counts

Although the previous figures indicate good results, such as the reduction in the number of cache misses, they do not show the most important result. The most important result is the impact on execution time. Execution time provides a clear, simple, unambiguous result. **Figure 11**.[1] and **Figure 12**[2] presents these "bottom line" results. They show the reduction in execution time within the speculative execution regions. Unfortunately, the bottom line results are not as impressive as the previous ones.

With the ORB prefetch enabled, on a number of workloads, execution time in the region increased. *compress95* and *art* exhibited extremely bad results. In general, the universal prefetch technique offered small improvements in execution time. However, the ORB prefetch technique frequently exhibited detrimental behavior. The only notable improvement was to the **m88ksim** workload. In addition, the combined technique showed a corresponding additive improvement in execution time.

There are a number of possible reasons as to why performance results were generally diluted (when transitioning from the number of locality misses to execution time). The first factor is the significance of data cache misses in relation to execution time in the regions. For example, if an application consumes only 10% of its cycle time waiting for data cache misses, then we are optimizing that 10% window of opportunity. In fact, *twolf* has this exact property. The 30% to 40% reduction in the number of cache misses (**Figure 9**) applies to only 10% of the region execution time. This results in a 3% to 4% improvement in execution time.

Other reasons for the generally poor performance results include perhaps (1) increased number of speculation violations, which was checked and proved not to be the case, (2) increased cache access latencies due to excessive traffic between the

[1] Compress95's ORB and Combination cycle reduction is actually -126%
[2] art's ORB and Combination cycle reduction is actually -40%

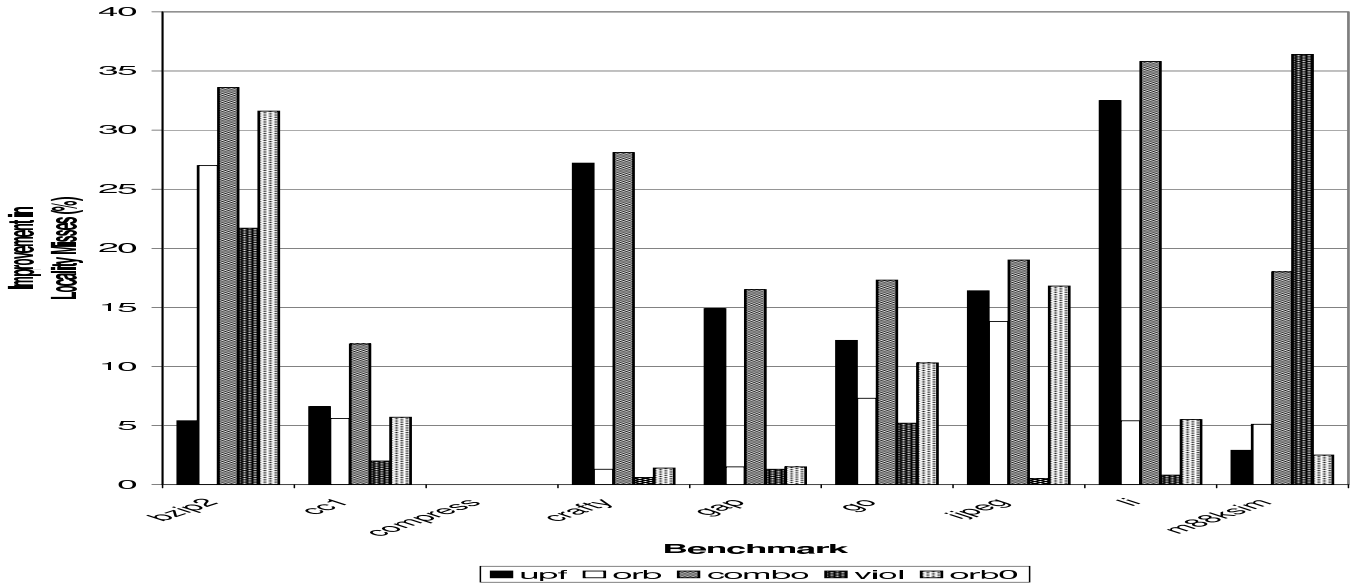

Figure 9: Improvement in locality misses.

caches, which was checked as well and turned out not to be the case, and (3) increased write invalidation coherence protocol costs, since a cache line may exist in all data caches due to our prefetching techniques. For the ORB prefetching technique, we noticed an increase in waiting for the home-free token phase. This is the most likely cause our the poor ORB prefetching results. Perhaps the ORB prefetch implementation has a number of flaws that lead to this problem. To verify whether the ORB prefetch was at all useful, we obtained results for an ORB prefetch implementation that had 0-latency prefetch times. The results shown in the graphs suggest that there are performance gains to be had.

5 Conclusion and Future Works

Based on the experimental results, we can conclude that locality conflict remains a open problem in TLS. Cold cache misses can be reduced by the use of universal prefetching. Universal prefetching can reduce cycles in the speculative region up to 22%. ORB prefetching shows promise in reducing the number of locality misses. It can reduce locality conflict by 30% in some cases. However, the proposed ORB prefetching technique increases communication cost and data latency significantly. The prefetching on speculative violation technique does not show significant promise. Even the most optimistic model shows less than 5% improvement in cycle counts. This is perhaps due to the fact that the number of speculative violations is rather small.

This project provides many valuable lessons. First, locality misses are not the only important parameters in determining system performance. Other factors include cache latency and communication cost. Second, cache reads and writes are two very distinct access patterns, and they have distinct characteristics. As such, one must consider each access pattern separately when designing cache optimization techniques.

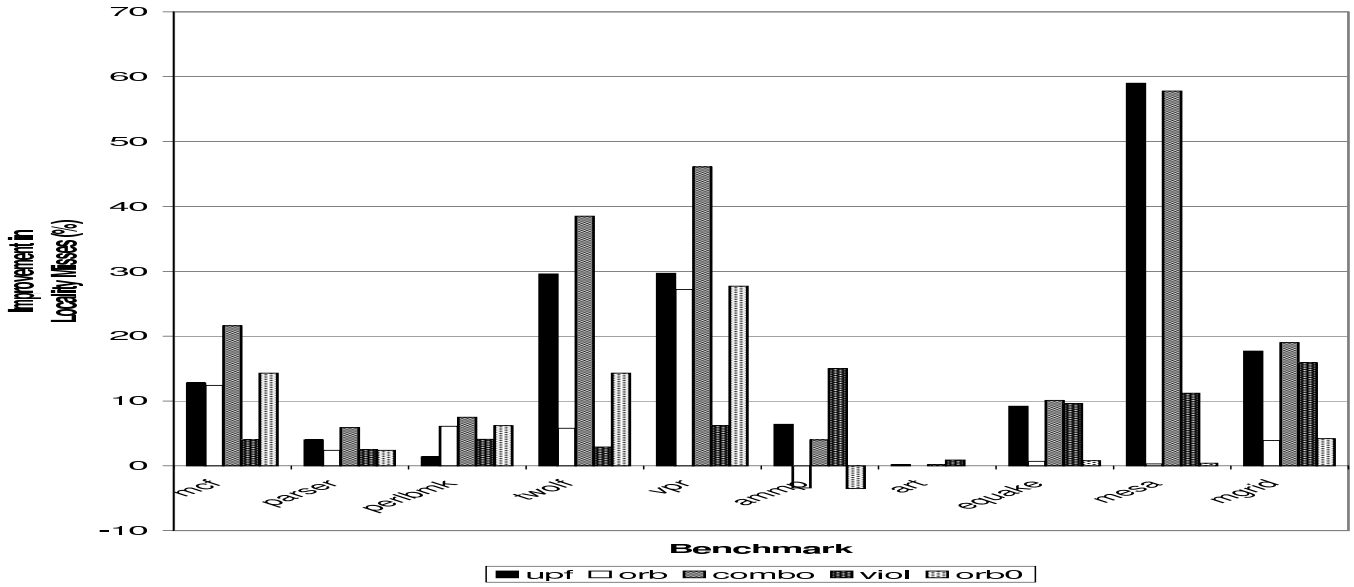

Figure 10: Improvement in locality misses(cont).

There are many tasks to complete as a subject of future work. First, the *more accurate* model for prefetching on violation needs to be completed. Next, the simulation code should be revisited to determine whether ORB prefetching will perform better through better modeling. There is, of course, more room to investigate how we could reduce locality conflicts when the data cache is dirty. Furthermore, improving the underlying cache/memory interconnection network may improve TLS performance.

References

[1] R. J. Figueiredo and J. Forles. Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocessors. In *Proceedings of International Conference on Parallel Processing*, September 2001.

[2] Jaekyuk Huh, Stephen W. Keckler, and Doug Burger. Exploring the design space of future CMPs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT2001)*, pages 199–210, September 2001.

[3] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, February 2002.

[4] M. Prvulovic, M. J. Garzaran, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of 28th Annual International Symposium on Computer Architecture*, pages 204–215, July 2001.

[5] Amir Roth and Gurindar S. Sohi. A quantitative framework for automated pre-execution thread selection. In *Proceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–442, November 2002.

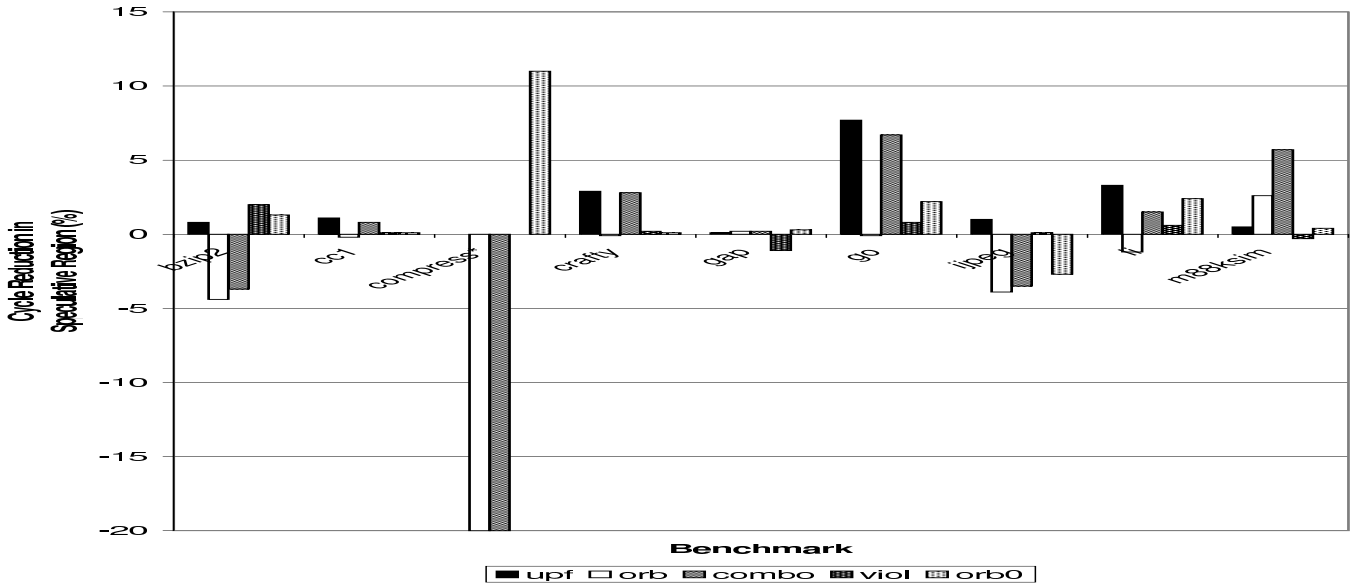

Figure 11: Improvement in cycles count

[6] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[7] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

A Appendix: Brief Description of Code

The code is modified from the Cello simulator, as provided by the instructor. The latest version is located in the EECG NFS directory /nfs/eecg/q/grads10/tamda/skule8/ece1718-hw/project/Simulators/cello/src. All of our modified code is enclosed by *#ifdef ECE1718* and *#endif* macros. The modification for universal prefetching is denoted by the macro *DUMB_PREFETCH* while the modification for ORB prefetching is denoted by the macro *ORB_NMRB_PREFETCH*. The modification for violation prefetch is denoted by the macro *VIOLATE_SPM_PREFETCH_ZLAT*. The macro *VIOLATE_SPM_PREFETCH* attempts to provide a more accurate model on violation prefetch.

The code for handling violation prefetch is in the method line_state::violate(). We model the most optimistic case by switching the state to shared (instead of invalid). Due to time constraint, we have not implemented and tested the *more* accurate model.

The implementation for handling ORB prefetch is as follows. In the method run_dref_handler() of memory_system class, the unified cache will be called to do dumb_prefetch(). The dumb_prefetch() will, in turn, tell all the upper objects (data caches) to prefetch_reference1(). The prefetch_reference1() method will call do_processor_action_coherence() to simulate the reading

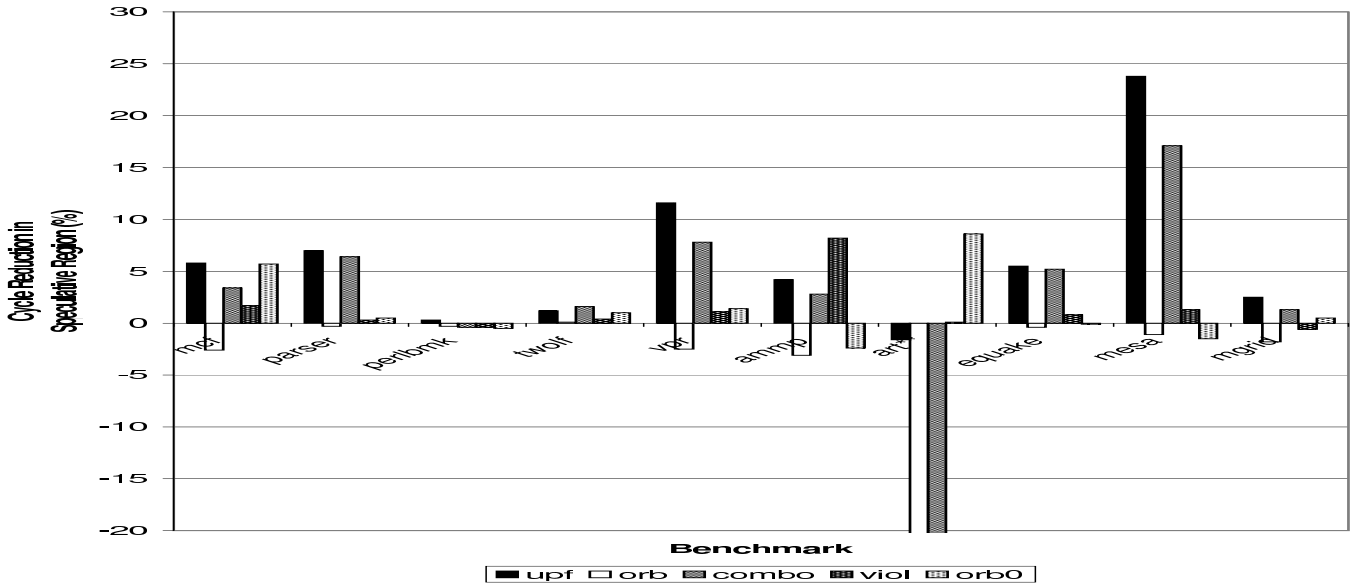

Figure 12: Improvement in cycles count.

of the cache line. The macro *NMORB_NOMODEL* models the most optimistic case by not modeling the communication cost for prefetching.

The implementation for handling universal prefetch is as follows. When the data cache tries to read speculatively and the line is in invalid state, it will send the request MS_ref_type_PREFETCH to the lower object. When the unified cache receives the prefetch request, it will check the state of the cache line. If the cache line is a good candidate for prefetching, it will send requests to all its upper objects for prefetching. In the method data_cache::do_request_from_lower(), the data cache will simulate reading the cache line from the unified cache (similar to the load_reference() call) when it receives a prefetch request.