CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms

Ravi Iyer
Communications Technology Lab, Intel Corporation
Hillsboro, Oregon
ravishankar.iyer@intel.com

ABSTRACT

Cache hierarchies have been traditionally designed for usage by a single application, thread or core. As multi-threaded (MT) and multi-core (CMP) platform architectures emerge and their workloads range from single-threaded and multithreaded applications to complex virtual machines (VMs), a shared cache resource will be consumed by these different entities generating heterogeneous memory access streams exhibiting different locality properties and varying memory sensitivity. As a result, conventional cache management approaches that treat all memory accesses equally are bound to result in inefficient space utilization and poor performance even for applications with good locality properties. To address this problem, this paper presents a new cache management framework (CQoS) that (1) recognizes the heterogeneity in memory access streams, (2) introduces the notion of QoS to handle the varying degrees of locality and latency sensitivity and (3) assigns and enforces priorities to streams based on latency sensitivity, locality degree and application performance needs. To achieve this, we propose CQoS options for priority classification, priority assignment and priority enforcement. We briefly describe CQoS priority classification and assignment options -- ranging from user-driven and developer-driven to compiler-detected and flow-based approaches. Our focus in this paper is on CQoS mechanisms for priority enforcement -- these include (1) selective cache allocation, (2) static/dynamic set partitioning and (3) heterogeneous cache regions. We discuss the architectural design and implementation complexity of these CQoS options. To evaluate the performance trade-offs for these options, we have modeled these CQoS options in a cache simulator and evaluated their performance in CMP platforms running network-intensive server workloads. Our simulation results show the effectiveness of our proposed options and make the case for CQoS in future multi-threaded/multi-core platforms since it improves shared cache efficiency and increases overall system performance as a result.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – *Cache Memories.*

General Terms: Algorithms, Performance, Design

Keywords: Cache, QoS, CMP, sharing, partitioning, performance.

1. INTRODUCTION

The efficiency and performance of caches is critical to the performance of microprocessors and platforms. This is especially true with the increasing gap between CPU speed and memory latency (ala memory wall [34]). To improve the efficiency of caches, researchers have proposed better cache organizations [3, 9, 23], better allocation and replacement techniques [2, 8, 17] and improved caching protocols [12, 16, 27]. Most of these techniques have been proposed in the conventional platform context, where in each cache was dedicated to a single CPU running a single thread at any given point in time. On the other hand, innovations in multithreading (SMT [31], HT [15], etc) and system-on-a-chip (SoC) or single-chip multiprocessors (CMP) are changing the nature of platform architecture and execution behavior. In these rapidly emerging architectures, the use of shared caches at some level in the cache hierarchy is desired due to its performance characteristics and design effectiveness [19]. In this paper, our focus is on improving the performance and efficiency of shared caches in multi-core (CMP) architectures. In general, the approaches discussed in this paper are applicable to caches in any platform that are shared by multiple memory access streams or flows from different threads, cores or devices.

In CMP platforms, shared caches are highly effective if the request streams are actually shared the data in the caches or have small working sets that collectively fit within the cache. Given the various workloads (web services, e-commerce, OLTP, financial applications, etc) that run on these platforms, this is difficult to guarantee. Furthermore, with the emergence of hosting services, utility computing and virtual machines, the CPU cores that are sharing the cache in the processor are like to have different applications running on them at any given time. The applications are likely to possess very different memory access characteristics and locality properties. As a result, conventional cache management approaches that treat all memory accesses equally are bound to result in inefficient space utilization and poor performance even for applications with good locality properties. To allow for the cache space to be utilized more effectively, we present a case for considering quality of service (QoS) in shared caches. Our proposed cache framework (CQoS) aims to improve shared cache efficiency by provide prioritized service to multiple heterogeneous threads sharing a cache structure.

To enable prioritization in shared cache structures, CQoS consists of mechanisms for priority assignment and priority enforcement. The first step however (before assignment and enforcement) is priority classification i.e. recognizing the heterogeneity in memory access streams and classifying them into the priority levels supported by CQoS. In this paper, we briefly touch upon the priority classification and assignment mechanisms that range from approaches that are user-driven and developer-driven to those that are compiler-detected and flow-based approaches. The focus of this paper is primarily on the CQoS priority enforcement mechanisms. Our proposed mechanisms for priority enforcement are (1) selective cache allocation, (2) static/dynamic set partitioning and (3) heterogeneous cache regions. We discuss the design trade-offs and implementation details of these CQoS techniques. Through extensive cache simulations of important usage scenarios in CMP platforms, we present the effectiveness of these mechanisms.

The rest of this paper is organized as follows. Section II presents a background on cache performance and motivates the need for CQoS. Section III introduces our CQoS framework for shared cache management. Section III also describes our proposed CQoS mechanisms for priority classification, assignment and enforcement. Section IV presents the design and implementation aspects of CQoS priority enforcement mechanisms options. Section V presents our simulation methodology for CQoS evaluation. Section VI analyzes CQoS performance in various scenarios and discusses the benefits of prioritization in cache space management. Finally, Section VII summarizes and concludes the paper with direction for future work in this area.

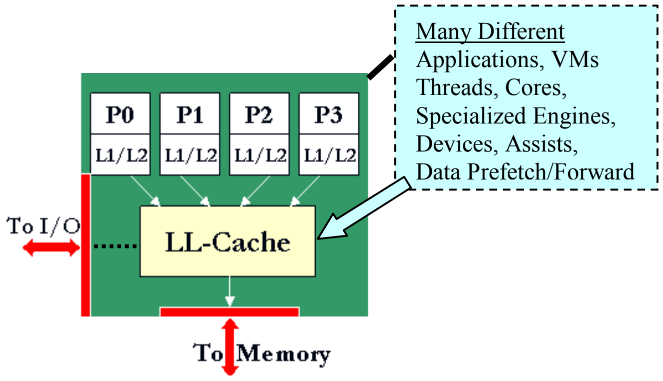

Figure 1. Shared Cache in CMP Processors

2. The Need For QoS in Caches

The typical architecture of a processor in a CMP platform is shown in Figure 1. As shown in the figure, a CMP microprocessor typically consists of a number of compute cores with individual L1 (and perhaps L2 caches). While the last level cache (LLC) can be made up of private caches per processor, studies [19] have shown that shared caching is more desirable from a performance and design point of view. Our focus is on the performance of this last-level cache as it is critical shared resource that is intended to keep the compute cores busy executing and the last line of defense against the memory wall.

Conventional cache management relies on the cache being used by only one memory access stream at any given point in time. However, as processors, systems and applications become far more complex, we need to consider the different types of memory access streams that allocate data into the shared LLC. Let us consider the potential for different memory access streams as can be broadly classified into the following categories:

(a) Multi-Threaded Applications: The simplest case is where the last-level cache is being used by multiple threads of the same application. In this scenario, if the threads are sharing and communicating a lot of data between each other, the conventional management of shared caches will work reasonably well. However, if the threads perform entirely different types of transactions concurrently (for instance – HTTP transactions in a web server application), then the performance of the concurrent transactions is dependent not only on its own locality, but also the nature of memory accesses generated by the other transactions. In such scenarios, the transactions that are of higher importance (e.g. secure payment transactions) should be prioritized higher than those of lower importance (e.g. browsing transactions).

(b) Multiple Heterogeneous Applications: When multi-tasking multiple applications in a CMP platform, it is likely that threads of one application and another get scheduled on to the same microprocessor, thereby sharing the last-level cache. This computing model is particularly gaining relevance/importance as virtual machines [5, 10, 32, 33] start to proliferate in data centers as a mechanism to reduce server sprawl. In such scenarios, different applications will definitely exhibit different memory access properties and therefore should be handled differently in terms of cache space allocated. The notion of cache space prioritization between is important here for high efficiency.

(c) Specialized Cores in CMPs: As application and network processing tends to frequently execute some common communication layers (TCP/IP, SSL for instance) or computing components (data encryption, compression, CRC, XML parsing, etc), architects are considering replacing one or two of the CPU cores with specialized cores for such components. In such scenarios, the processing and memory access flows generated by the cores on the CMP will definitely have different properties and can be best handled with that knowledge.

(d) Sharing Caches between Cores and Devices: With the imminent potential of computing appearing in I/O devices (and possibly management controllers) in the system and the integration of I/O links into the CPU, researchers and architects are evaluating the benefits of using cache space to speed up the processing on the device. In such scenarios, it becomes important to perhaps partition the cache space dynamically between the cores and the devices.

(e) Memory Latency Helpers: In current processors, prefetching is employed to overlap computation with memory access. Excessive prefetching [22, 30] is known to cause problems such as cache pollution and deteriorate application performance. In addition to prefetching [25] initiated by the CPU, researchers have considered memory-side prefetching, data forwarding [1, 11, 21] and direct placement of network data into CPU cache [18]. To reduce the amount of pollution caused by these memory latency helpers, it is important to prioritize cache space usage between demand activity and prefetching/forwarding activity.

Several other considerations need to be kept in mind when considering prioritizing cache space utilization. Other than the basic priority of the application, it also needs to be kept in mind that providing higher cache space to a higher priority application does not always guarantee higher cache performance. This depends heavily on the application's memory access characteristics (i.e. locality properties). As a result, locality and "user-defined" application priority need to be considered in unison to form the cache priority of the memory access stream. Other aspects to consider are the dynamic changes to the priority of an application or memory access stream. Since processing tends to go through several phases, it is important that the priority assignment mechanism be designed to allow dynamic changes. These issues motivated us to develop a framework that introduced the notion of quality of service in caches. In the next section, we will discuss the basic ideas behind this and present several potential design and implementation options to enable this in future systems.

3. CQOS -- IMPROVING SHARED CACHES

As motivated in the previous section, it is important to introduce prioritization in order to provide the notion of quality of service in shared cache space provisioning. In this section, we introduce the CQoS framework to enable this. Before we do that, we take a brief look at the performance implications of conventional cache management. Using a web server workload (a SPECweb99 trace) and a network-intensive workload (a NTttcp trace), we show the cache performance when the workloads run independently in the cache and when they are run simultaneously through the cache. Figure 2 shows the impact of conventional cache management in these scenarios.

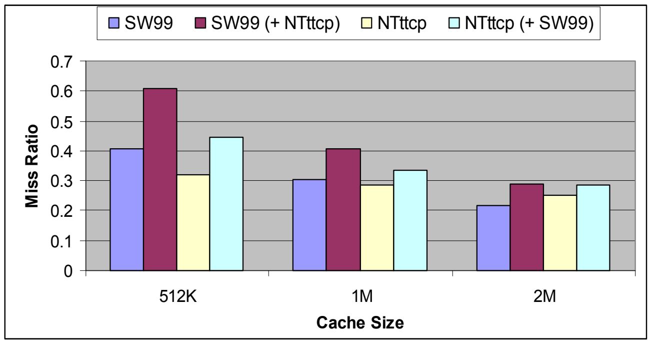

Figure 2. Issues w/ Conventional Cache Management

Several observations can be made from the results in Figure 2. First of all as expected, when the workloads are run independently, the improvement (reduction) in miss rate with cache size is significantly better for SPECweb99 than for NTttcp (due to their locality characteristics). Secondly, there is a significant increase in miss rate for both workloads when run simultaneously (increase of 35-40% for SPECweb99 and 14-40% for NTttcp). Third, in this case, an efficient shared cache management approach would be to improve SPECweb99 (the application with more locality and more priority from the user's point of view). The ultimate goal is to provide SPECweb99 with a miss rate that is at the same level as when it is run independently, while not degrading the performance of NTttcp significantly.

3.1 CQoS: Basic Framework and Flow

In order to provide more cache space to higher priority and high locality memory access streams, we introduce the CQoS framework. The CQoS framework is made up of three aspects to managing cache priorities for memory access streams:

[1] Cache Priority Classification: Here, the emphasis is on identifying the heterogeneous memory access streams and classifying them into priority levels. These priority levels may be different from the number of priority levels available in the platform. To start with, we first define the required levels of priority by taking an in-depth look at the different memory access streams described in Section II earlier.

[2] Cache Priority Assignment: Once the workload's priority levels have been determined, these will be translated into the system cache's priority levels. As will be discussed in a later subsection, the options for cache priority assignment range from being ISA-based, memory type-based or flow-based and these may be utilized either by the compiler, the developer, user or the O/S scheduler.

[3] Cache Priority Enforcement: Once the memory accesses are classified and assigned into priorities, the focus here is to enforce these priorities during cache space allocation and management. This can be done by monitoring the space consumption for each priority level and modifying either the allocation process or the replacement process. Alternatively it can also be accomplished by structuring the single shared cache as multiple smaller caches that are organized differently and have different policies enforced. The specific techniques for cache priority enforcement will be discussed in significant detail in a later subsection.

Due to space limitations, it should be kept in mind that we will introduce the priority classification and assignment approaches in this paper, but focus more heavily on the design and implementation of the options for cache priority enforcement.

3.2 CQoS Priority Classification Options

As mentioned above, the first step for CQoS is to recognize the heterogeneous memory accesses involved and classify the memory access stream or data structure within the stream into priority levels. This can be achieved in the following ways:

[1] Based on Data Structures or Access: The types of data structures touched in typical memory access streams during any processing can be classified into three types: (1) frequent or hot-set and (2) typical or average-set and (3) one-touch or cold-set. If the application developer or compiler can profile the application and identify these, then they can be taken advantage of in two ways: (1) load /store instructions tagged with the above attribute or (2) the data can be allocated in different memory regions with specific attributes. Once tagged, these can be handled differently in the cache based on the priority level.

[2] Based on Transactions or Phases: Another approach is for the developer to define priorities for each phase or transaction processed by the application. In this way, whenever the phase is entered, the cache controller can be informed about the priority level of the subsequent memory access stream and handle the memory access appropriately. Upon exit or entrance into another phase, the priority level can be modified.

[3] Based on Flow Type: This approach requires the hardware to differentiate between demand memory accesses and prefetch accesses made either by the core or other components in the system. Since this information is mostly available in the system, this probably does not require additional support.

[4] Based on Different Applications: If applications as a whole are largely homogeneous, they can be classified at a certain priority level. In this case, this application priority has to be maintained along with the process or thread context information in the system and made available to the cache controller.

[5] Based on Types of Cores Devices or Threads: Finally, this approach requires the cache to differentiate between different cores or devices making requests. For instance, requests coming from an I/O device may be classified at a lower priority level than the requests coming from the processor core. Similarly, a specialized core assisting the application may also be classified at a different priority level than the main application core. In such scenarios, the BIOS or the O/S needs to configure each device with a certain priority level which may or may not be modified by the system administrator or system management controller.

While it would be great if support for all of the above options were available in the platform, we expect that some trade-off analysis is needed to understand the feasibility of the options proposed. The above options vary in their granularity and the hardware/software support needed for the platform. It should also be noted that two or more options can be combined to compute the overall priority level of the memory access stream. Once the priority level is decided upon, the next step is identify the mechanisms for priority assignment and map these memory access priority levels to priority levels made available by the platform.

3.3 CQoS Priority Assignment Options

In the previous section, we discussed the different methods by which memory accesses can be classified into priority levels and also alluded to the potential mechanisms in which they can be translated into priority assignment for the cache controller. In this section, we present these directly and understand how they relate to the compiler, developer, user, O/S scheduler, system administrator. The mechanisms include:

[1] Tagged Memory References (Loads/Stores): It may be possible to provide different load instructions for each priority level supported by the cache controller. It should be noted, however, that in order to use the available opcodes minimally, it is probably feasible to support only two or three in the instruction set architecture (ISA). Similarities to this can be found by looking at the prefetch instructions for Intel's IA-32 processor family. The ISA for the Intel IA-32 supports three different types of prefetches: (1) for temporal prefetches that bring data into L1 & L2, (2) for temporal prefetches that bring data into just the L2 and (3) non-temporal prefetches that place the data into only 1-way of the L2 cache. The three prefetch types map directly to data types with different locality properties and therefore different priorities.

[2] Priority Specification Instructions: This mechanism basically requires ISA support for executing an instruction that causes the process / thread / core to be assigned a certain priority level. The priority level specified by the instruction is recorded in a register or a table (in case of a multi-threaded core) and can be provided to the cache controller when subsequent loads and stores are generated.

[3] Priorities in Memory Types: Today, there are four or five dominant memory types available in a typical platform (e.g. write-back, write-combining, un-cacheable and write-protected). These memory ranges are available through range registers in order for the cache and chipset to identify the type of load/store access that was generated. If these memory types can be further segregated into low and high priority regions and appropriate memory allocation system calls are made available in the O/S, then the application developer (or perhaps the compiler) can use the priority level of the allocated memory region and translate it to an appropriate allocation during runtime.

[4] Application Priorities: Today, applications can be assigned priorities by the user in many O/S'es (Linux, Unix, HP-UX, etc) at the process granularity. However, this priority level is used for the scheduler to provide more or less time intervals to the application. It would be useful to add a component priority level (such as cache priority level) to the application in order for it to be maintained in the process's context and be accessible to the application and the O/S if it chose to use it for cache priority assignment. This essentially adds context overhead to the running process which needs to be saved and restored when the process is swapped out.

[5] Device / Core Based: This is essentially the hardware designer (through BIOS) or the O/S developer assigning priorities to devices in the system and providing the priority level so that all accesses made by the entity is treated with the appropriate level of priority.

3.4 CQoS Priority Enforcement Options

The focus of this paper is largely on the design, implementation and performance of CQoS priority enforcement. Given a priority level for each memory access, the problem statement is essentially that of enforcing it in cache space management. It should be noted that we are only discussing quality of service in terms of cache space provided and not in terms of cache latency or miss ratio since these may be entirely dependent on the inherent characteristics of the application. To enforce the priority levels in the cache, we propose the following three different types of mechanisms:

[1] Set Partitioning Schemes (Static/Dynamic): Caches are organized in sets containing one or more lines. The goal of this mechanism is to allow higher priority applications to occupy more ways of the set and lower priority applications fewer ways in the set. This can be achieved by statically decomposing the set into a number of subsets or dynamically placing a limit on the number of lines occupied (within the set) by the memory access streams at a given priority level.

[2] Selective Cache Allocation: This mechanism does not disturb the organization of the cache but maintains counts to determine the number of lines occupied in the cache by memory access streams at a given priority level. Based on the count, it probabilistically allocates or rejects cache line allocation requests made to the cache.

[3] Heterogeneous Cache Regions: This mechanism proposes heterogeneous cache structures (set-associative caches, stream buffers, victim caches) that can be mapped to memory access streams based on the priority levels. Alternatively, it also discusses heterogeneous cache regions with different replacement policies (e.g. locked or self-invalidated lines).

A discussion on the design and implementation options of these priority enforcement schemes will be presented in the next section.

4. DESIGNING PRIORITY ENFORCEMENT OPTIONS

In order to make the design and implementation feasible, we largely focus on enabling two priority levels in each mechanism. Many of the mechanisms proposed in the previous subsection, however, are easily extendable to supporting multiple priority levels.

4.1 Design of Set Partitioning Techniques

In order to enable the partitioning of the shared cache into different spaces for different priority levels, we further decompose the set down into subsets. In this section, we discuss the design of a static and dynamic set partitioning approach to manage subsets. We start with static set partitioning.

4.1.1 Static Set Partitioning Techniques

We start with each set containing N cache lines in the shared cache. To statically separate the set into two subsets of different sizes, we logically map the first X lines to priority level 2 and the next N-X lines to priority level 1. It should be noted that we use lower values to indicate higher priority (i.e. priority level 1 being the highest). The changes required to support this mapping and partitioning are as follows:

– **Cache Lookup:** All lookups in the cache will scan through all the lines in the set.
– **Cache Allocation:** If the line does not exist in the cache, then the allocation is done to the subset that the priority level is mapped to.
– **Cache Replacement:** When a victim has to be chosen, it is chosen from the subset that the priority level is mapped to.

The implementation cost of this scheme is in the changes to the bits maintained for replacement purposes and to the replacement mechanism itself. The replacement mechanism basically needs to be modified to cater to multiple different subsets. An optimization to this scheme is to allow the highest priority scheme to allocate anywhere in the set, but the lower priority schemes to allocate only in the subset that they are mapped to. This way, the highest priority scheme can always utilize the entire cache space even when there are no applications in other priority levels.

4.1.2 Dynamic Set Partitioning Techniques

As before, we start with each set containing N cache lines in the shared cache. In static set partitioning, we physically partitioned the set into two subsets. In dynamic set partitioning, the approach is to achieve the partitioning by imposing a limit on the number of cache lines that can be occupied in the set by a given priority level. For instance, priority level 2 can occupy only X lines in the cache, whereas priority level 1 can occupy up to N lines in the set. The changes required to achieve this partitioning are as follows:

– **Cache Lookup:** All lookups in the cache will scan through all the lines in the set.
– **Cache Allocation:** If the line does not exist in the cache, then the allocation is done anywhere in the set as long as the set limit has not been reached for the priority level of the allocation request. If the limit has been reached, the replacement scheme below is used.
– **Cache Replacement:** When the set limit is reached for a given priority level, then the replacement policy needs to locate the lines in the cache that are associated to that priority level and choose a victim among those lines.

The implementation cost of this scheme is the following: (1) maintaining a count per set per priority level, (2) maintaining the id of the priority level for each line in the set and (3) changes to the bits maintained for replacement purposes and to the replacement mechanism itself. The replacement mechanism basically needs to be modified to cater to multiple different lines when the set limit is reached. In addition, it also needs to be able to locate the lines associated with the priority level before choosing the victim among those.

The performance comparison between static and dynamic set partitioning as well as an analysis of the overall benefits of set partitioning will be presented in the next section. Qualitatively, we expect dynamic partitioning to provide more efficiency in the cache since it allows at least one memory access stream at the highest priority level to occupy the entire cache. These schemes can also be generalized to support P different priority levels. However, this is not within the scope of this paper (due to space limitations).

4.2 Design of Selective Allocation Techniques

Another approach to providing different amounts of cache space to different priority levels is to monitor the allocation amounts in the overall cache and allow / disallow allocation based on the current space utilization. In this section, we introduce a simple scheme called probabilistic cache allocation.

4.2.1 Probabilistic Cache Allocation

In order to limit the cache space utilized by a given priority level, we use an allocation probability (AP). The allocation probability (AP) implies that the streams at that priority level can only occupy up to AP*C bytes in the cache, where is the size of the cache in bytes. Once each priority level is assigned an allocation probability, the subsequent requests generated by memory access streams of that priority level are made to adhere to that allocation amount probabilistically. This is more easily understood with an example. Let us assume two priority levels with allocation probabilities assigned as AP2 = 30% for priority level 2 and AP1 = 70% for priority level 1. In this scenario, when a request I is generated in a memory access stream, a random number of generated between 0 and 100. The random number (RI) is compared against APx where x is the priority level of the stream. If RI <= APx, then the cache line is allocated into the cache. If RI > APx, then the cache line is dropped for allocation. Instead of dropping the cache line entirely, an extension to this scheme is to place the cache line in a victim cache and consider it for allocation again if the line is touched by the processor.

For this scheme, the implementation cost is essentially in maintaining the allocation probabilities per priority level and generating a random number for each memory access stream. The only change in cache management is to the cache allocation policy, where the probabilistic allocation is performed as opposed to direct allocation.

4.3 Design of Heterogeneous Cache Regions

In this class of techniques, we basically propose two approaches to maintaining heterogeneous cache regions: (1) heterogeneous caches mapped to priority levels and (2) heterogeneous cache line policies mapped to priority levels. These schemes are described further below.

4.3.1 Heterogeneous Caches

Here, the basic idea is to provide multiple caches with different organization structures or policies. For instance, one simple heterogeneous cache structure could separate a 4MB, 8-way shared cache with 128-byte lines into three different cache structures: (1) a 12-way 3MB cache structure with 64-byte lines, (2) a direct-mapped 512K cache with 32 byte lines and (3) a 512K FIFO stream buffer with 1K cache lines. The basic premise behind this separation would be the presence of three different types of memory accesses -- (1) Transient data that is streamed through the cache has spatial locality but almost no temporal locality (as is the case for network processing and graphics applications), (2) Temporal data with little spatial locality and (3) Data types with average temporal and spatial locality.

In this scenario, the mapping of priority of memory types to these caches needs to be done with multiple aspects in mind – (1) the size requirements of the working set and (2) the temporal/spatial locality properties of the memory access type. Depending on these two aspects, it is easy to determine the right cache to map the priority level to. A concern that remains is the fragmentation issue – where some cache spaces are rarely used and could have been put to better use if they were available. This remains a limitation of this scheme and can be perhaps avoided in the approach discussed below.

4.3.2 Heterogeneous Cache Lines

Here, the idea is to separate the property of the cache line from the policies enforced on the cache. For instances, cache lines typically fall into one of the following categories: (a) frequently accessed or hot-set, (b) typical or average-set, (c) one-touch or cold-set. However, due to conflicts in the cache, capacity limitations and low visibility into the actions on the lower level caches, these properties cannot be held using the typical LRU policy in the last-level cache. However, if the priorities were mapped according to the type of the memory access, then the policy used to manage that line in the cache can be changed.

Essentially, we are proposing the use of the following two types of cache line management approaches:

(1) Cache Line Locking: By allowing cache lines to be excluded from the cache replacement policy, the line belonging to a hot-set can be locked into the cache. Given the typical workload, most of the accesses occur to less than 20% of the lines. In such cases, this policy could help improve the application performance tremendously.

(2) Selective Self-Invalidation: By allowing cache lines to be invalidated in the cache, the memory access types that fall into the cold-set category can be easily self-invalidated after access. Support for self-invalidation of cache lines is available in some microprocessors today via a instruction. Here, we propose to do self-invalidation in hardware based on the priority level of the cache line that is accessed.

5. EVALUATION METHODOLOGY AND TOOLS

In this section, we describe our trace-driven simulation methodology to understand the cache performance benefits of CQoS priority enforcement schemes.

5.1 Workloads and Traces

We collected traces from two network-intensive workloads (SPECweb99 and NTttcp) running on current platforms in our lab. The workloads can be described as follows.

– **SPECweb99** [26] is a benchmark that attempts to mimic a web server environment. The benchmark setup uses multiple client systems to generate aggregate load on the system under test (a web server). Each client (mimicking browsers) initiates TCP connections to the web server and makes HTTP requests for static or dynamic web pages. SPECweb99 requires 30% of the requests to be dynamic requests and 70% to be static requests. SPECweb99 also uses popular file access characteristics (Zipf distribution over directories and files) and persistent connections to represent current web server accesses.

– **NTttcp** [29] is Microsoft's command-line sockets based tool based on the ttcp benchmark, which is used for measuring TCP and UDP performance between two end-systems. NTttcp achieves high performance by filling a memory buffer with data, then repeatedly transmitting this data. For our analysis of end system performance, the traces were collected on the server where the NTttcp receiver resided.

5.2 Cache Simulation Methodology

Our evaluation methodology consists of an extensive set of cache simulations fed by traces collected on a current platform running SPECweb99 and NTttcp, as described in the previous section. We then extracted the memory reference streams from these traces and fed those through cache simulation models developed using our CASPER (Cache Architecture Simulation and Performance Exploration using Refstreams) simulation environment [6, 7]. CASPER provides a rich set of features for detailed cache evaluation studies such as the following:

– UP Cache Hierarchies -- unified & split I/D caches

– MP Cache Hierarchies -- MESI & broadcast-based

– CMP Cache Simulations -- multiple cores or devices with individual caches or shared caches

To study the performance implications of CQoS priority enforcement schemes, we simulated a shared cache between processor cores and I/O devices. We modeled three different types of CQoS mechanisms in CASPER -- static set partitioning, dynamic set partitioning and selective cache allocation. We also evaluated heterogeneous cache regions by extending the cache simulation model within SimpleScalar [24]. This last study did not use CASPER since we already had the necessary support built into SimpleScalar for a different investigation [35].

6. CQOS PERFORMANCE CASE STUDIES

In this section, we describe the simulation benefits of CQoS by picking three different scenarios – (1) multiple applications sharing the cache, (2) multiple devices sharing the cache and (3) a single application possessing distinct data structures with differing memory access properties.

6.1 Impact of CQoS on Multiple Applications

To study the impact of CQoS on two applications sharing the cache in a CMP platform, we ran SPECweb99 and NTttcp traces through a shared cache ranging from 512K to 4M in size. We chose relatively small cache sizes since the working sets of these applications are small due to the fact that they are running at today's performance levels.

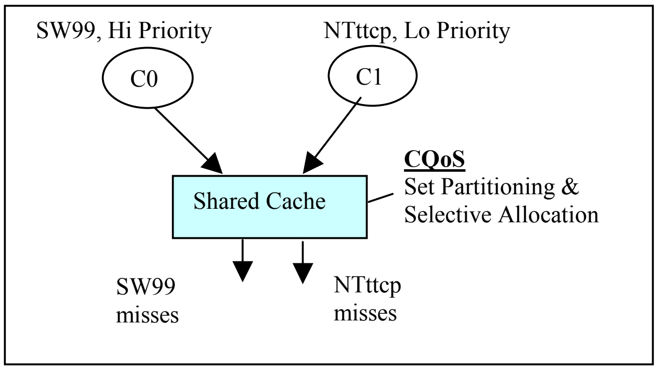

Figure 3. CQoS Study on Multiple Applications

Before we show the impact of CQoS on cache performance, we present the impact of conventional cache management on SW99 performance with and without NTttcp running through the same shared cache. Figure 4 shows the results from this study. It should be noted that the aim of CQoS is basically to provide better performance for higher priority applications which comes at the expense of the lower priority application.

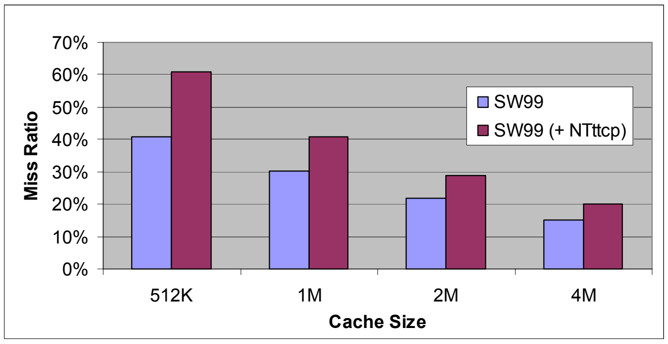

Figure 4. Conventional Cache Management

As shown in Figure 4, when using conventional cache management, the cache miss rate of SW99 increases by up to 40% when a low priority application (NTttcp) is also run through the same shared cache. This can be improved substantially if CQoS is used in the cache. We have tried both dynamic set partitioning and selective allocation to understand their impact. The results are shown in Figures 5 and 6.

Figure 5 shows the impact of imposing a limit of X lines on NTttcp in the shared cache. The value for X is varied from 0 to 4. The "No Limit" case depicts the scenario where CQoS set partitioning is not enforced. From the figure, we find that imposing a limit of 2 lines in an 8-way set occupied by NTttcp can improve the 2MB cache performance of SW99 greatly (a decrease of ~15% in miss rate). This does come at a cost to NTttcp (an increase of miss rate from 28% to 35%). However, it should also be noted that SPECweb99 is chosen to be the higher priority application and its performance is more critical as well as more sensitive to miss rate than NTttcp. Projecting overall performance for these workloads under CQoS is not within the scope of this paper.

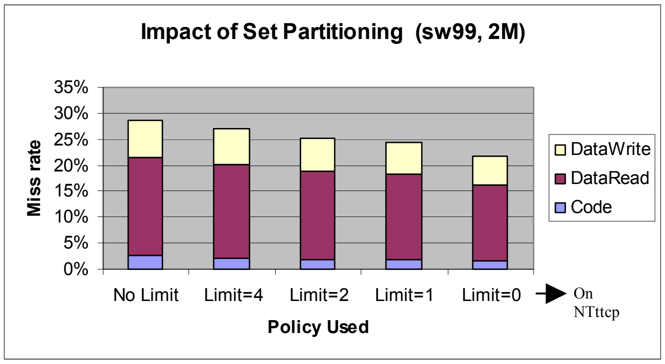

Figure 5. CQoS Set Partitioning

Figure 6 shows the impact of selective cache allocation on SW99 cache performance (with cache size of 2MB). The allocation probability for SPECweb99 was held at 100% whereas the allocation probability for NTttcp was varied from 0% to 50% (as denoted in the x-axis). The "No Alloc" entry shows the case where no CQoS priority enforcement was enabled. The results show a steady decrease in cache miss rate of SW99 as the allocation probability for NTttcp is reduced from 50% to 0%. An allocation probability of 25% (for NTttcp) reduces the SW99 cache miss rate from 29% to 26% (a reduction of 12%). The impact of the 25% allocation probability on NTttcp performance is that the miss ratio increases from 28% to 31%.

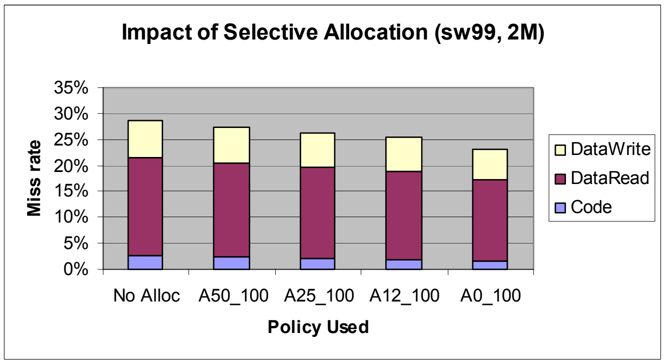

Figure 6. CQoS Selective Allocation

These preliminary studies confirm the benefits of CQoS cache priority enforcement in scenarios where there is clearly one high priority application and one low priority application.

6.2 Impact of CQoS on Multiple Devices

In this section, we study the cache performance of SPECweb99 in a dual-core platform with a high priority memory access stream running on the processor (P2M) interfered by a lower priority stream that is generated by the I/O device (IO2M).

We first varied the 8-way shared cache size from 2MB to 8MB. Each processor was enabled with an private cache of 1M. With conventional cache management, the IO2M stream interferes with the P2M stream significantly, causing the cache performance of the high priority stream to reduce considerably as shown in Figure

8. For example, the P2M miss ratio for a 8M cache increases from 14% to 23% when IO2M is allowed to use the shared cache.

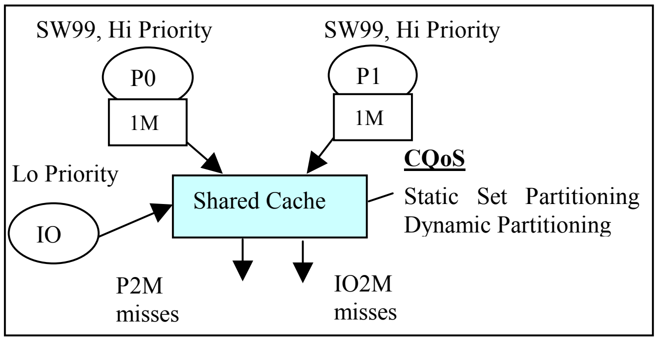

Figure 7. CQoS Selective Allocation

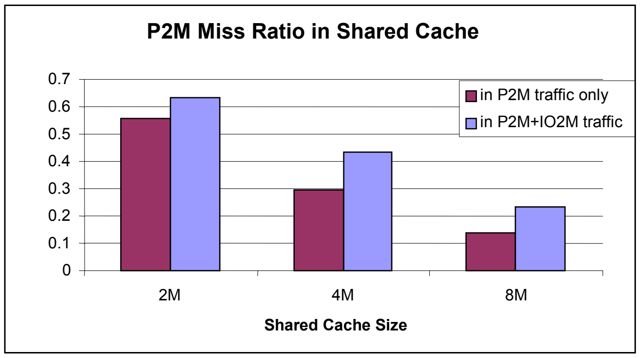

Figure 8. Conventional Cache Management

To show the benefits of CQoS, we chose the 8M cache to perform set partitioning studies and analyses. We started with the use of dynamic set partitioning by imposing a limit on the lines or ways occupied by the IO2M memory access stream. Figure 9 shows the cache miss ratio (for P2M traffic, for IO2M traffic and overall) as the IO2M limit is reduced from 8 (entire set) to 0 (no line in the set). To achieve best P2M traffic, it is obvious that limit of 0 on IO2M would be the most desirable. However, to allow some cache benefits for the IO2M traffic, it appears that allowing 1-way for IO2M traffic achieves good IO2M miss ratio as well as greatly reduces P2M miss ratio from the conventional cache scenario (from 23% to 18%).

Figure 10 shows the performance comparison of dynamic set partitioning to static set partitioning. From the figure, it can be noticed that static and dynamic set partitioning works to produce similar performance when the subset limits are 3 or below. However, when using static set partitioning, as the IO2M limit is increased beyond 3, the performance of P2M traffic suffers considerably because it is limited to the remaining few lines only. The hard partitioning impact of set partitioning should be kept in mind when considering static set partitioning beyond a small number of ways in the set.

6.3 Impact of CQoS on Specialized Cores

In order to understand the potential of CQoS on specialized cores, we have studied the potential for a dedicated core running the TCP/IP protocol processing [20, 4] for network-intensive servers [14]. In Figure 11, we show both the application and network cache accessing a shared cache organized as two different cache structures – Application cache and Network cache. In a previous subsection, we showed how application (SPECweb99) cache performance is hurt when running a network intensive workload (NTttcp) in a shared cache. Here we study the cache size requirements of the dedicated network cache and then split that further based on data types touched during TCP/IP processing (specifically receive-processing that is known to be memory intensive [13]).

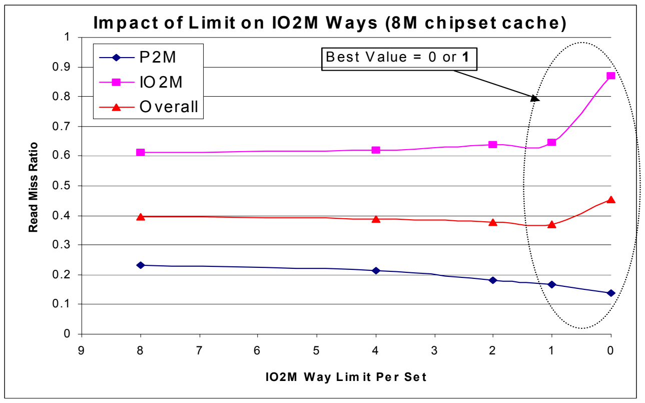

Figure 9. Impact of Dynamic Set Partitioning

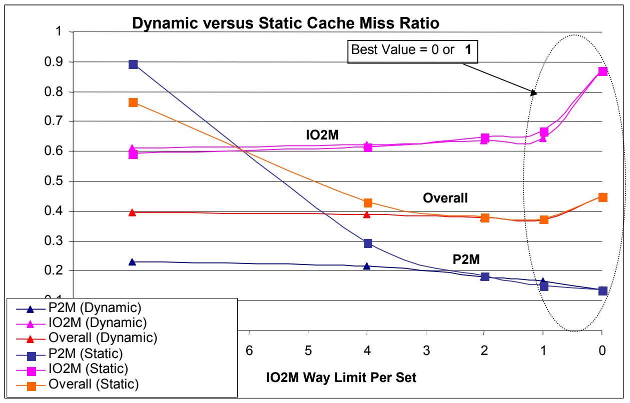

Figure 10. Dynamic versus Static Set Partitioning

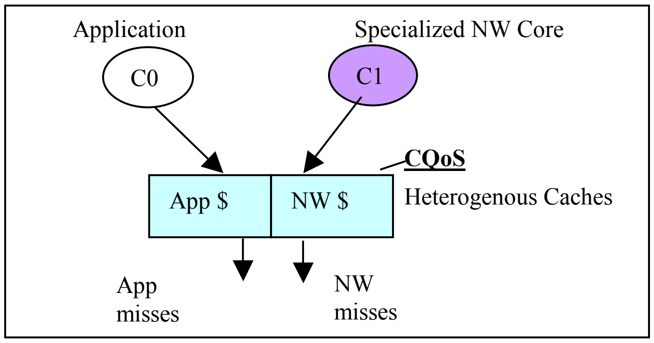

Figure 11. Dedicated Network Processing

Figure 12 shows the size requirements for a receive-intensive (RX) workload. As shown in the figure, a dedicated cache size of less than 32KB is sufficient for network processing. Providing additional cache space just pollutes the cache with non-temporal data. By investigating the data types touched, we have also identified two distinct types of data:

[1] Data with temporal locality: These include connection context information and hash tables that are touched when every packet is received. This also includes local variables that the stack uses for processing.

[2] Transient Data: These include incoming network data (descriptors, headers and payload) that have to be invalidated from the cache and are compulsory misses.

To further reduce the network cache size, we separated the cache into two heterogeneous structures – a set-associative temporal locality cache (TLC) and a FIFO stream buffer (SB). The TLC caches data that has temporal locality whereas the stream buffer holds the transient data. Figure 13 shows the comparison between the size requirements of TLC+SB versus that for the simple network cache.

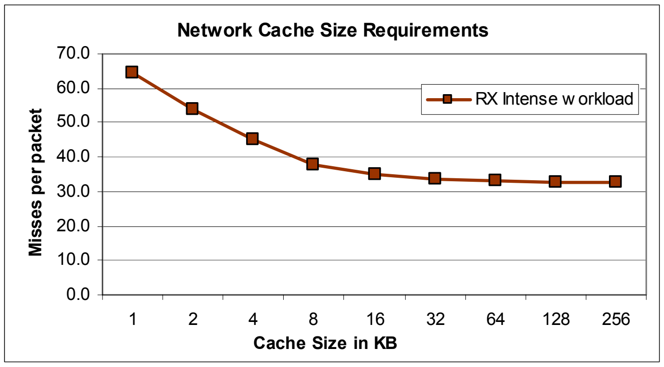

Figure 12. Network Cache Size Requirements

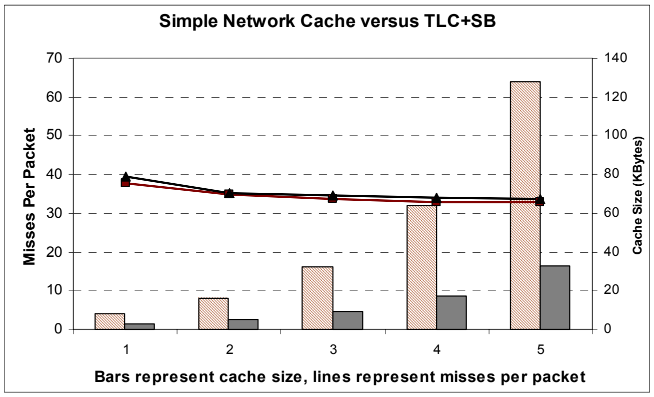

Figure 13. Benefits of Heterogeneous Cache Regions

As shown in Figure 13, the performance (lines illustrating misses per packet) is about the same between the simple network cache and the combination of TLC + stream buffer. However, the cache size required for TLC+SB is much lower (as represented by the vertical bars). The benefits of heterogeneous cache regions in this case materializes as cache size reduction especially since the logic required to maintain stream buffers is also rather low.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the notion of quality of service (QoS) in shared cache management (especially in CMP platforms). Our CQoS framework enabled QoS in shared caches by using priorities. The steps involved in CQoS were the following: (1) priority classification for each memory type or access stream, (2) priority assignment mechanisms for translating the stream priorities into those supported in hardware and (3) priority enforcement mechanisms. While we discussed several options for priority classification and assignment, our focus in this paper was on the design, implementation and performance evaluation of CQoS priority enforcement.

Specifically on CQoS priority enforcement, we proposed three different mechanisms: (1) static/dynamic set partitioning, (2) selective cache allocation and (3) heterogeneous cache regions. We then discussed the design trade-offs and implementation potential of these mechanisms. Finally, by modeling the options in a cache simulator, we studied the performance benefits of CQoS. Our preliminary evaluation was performed on network-intensive workloads where we find three major classes of different traffic: (1) CPU application traffic, (2) device I/O traffic and (3) network protocol processing traffic. By prioritizing application traffic over I/O device traffic and network processing traffic, we showed how the performance benefits can be achieved. Then by focusing on network processing, we showed that heterogeneous cache regions can help reduce the dedicated cache size needed for TCP/IP processing engines (specialized cores) and thereby reduce pollution for the application processing as well.

There is abundant work to be done in this area of research. To start with, we would like to perform our CQoS evaluation on several different types of applications. We also plan to perform an in-depth investigation into CQoS priority classification and assignment mechanisms – especially in using compilers for automating this process. We would also like to understand the tradeoffs in enabling new memory types and allocation primitives in the O/S for improved cache performance. We plan to build a hardware prototype to better showcase CQoS benefits between complex applications like virtual machines. Finally, we believe that this notion of prioritization is not only applicable to caches but also to other shared resources in the system (like shared interconnects, memory subsystem bandwidth etc). We believe that enabling QoS in the system as a whole will be an important contribution to future server platform architectures running a variety of applications.

8. REFERENCES

[1] H. Abdel-Shafi, et al., “An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors,” Proceedings of the 3rd International Symposium on High-Performance Computer Architecture, February 1997, 204-215.

[2] K. Beyls, “Faster Computing through Software-Controlled Cache Replacement,” http://escher.elis.ugent.be/publ/Edocs/DOC/P102_118.pdf

[3] F. Bodin, A. Seznec, ``Skewed Associativity improves performance and enhances predictability'', IEEE Transactions on Computers, May 1997

[4] D. Clark et. al., “An analysis of TCP Processing overhead”, IEEE Communications, June 1989.

[5] T. Garfinkel , Ben Pfaff , Jim Chow , Mendel Rosenblum , Dan Boneh, “Terra: a virtual machine-based platform for trusted computing,” Proceedings of the 9th ACM symposium on Operating Systems Principles, Oct 2003, NY, USA

[6] R. Iyer, “CASPER: Cache Architecture, Simulation and Performance Exploration using Re-streams,” Intel’s Design and Test Technology Conference (DTTC), 2001.

[7] R. Iyer, “On Modeling and Analyzing Cache Hierarchies using CASPER,” MASCOTS-11, 2003.

[8] P. Jain, et al., “Software Assisted Cache Replacement and Prefetching Pollution Control,” http://www.csail.mit.edu/research/abstracts/abstracts03/architecture/24jain.pdf

[9] N.P. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” Proceedings of 17th International Symposium on Computer Architecture, pages 364--373. IEEE, June 1990.

[10] S.T. King, George W. Dunlap, Peter M. Chen, "Operating System Support for Virtual Machines", Proceedings of the 2003 Annual USENIX Technical Conference, June 2003.

[11] D. Koufaty, et.al, “Data Forwarding in Scalable Shared Memory Multiprocessors, IEEE TPDS, 1997.

[12] D. Lilja and P-C. Yew, “Combining hardware and software cache coherence strategies,” International Conference on Supercomputing, 1991.

[13] S. Makineni and R. Iyer, “Architectural Characterization of TCP/IP Packet Processing on the Pentium® M microprocessor,” HPCA-10, 2004.

[14] S. Makineni and R. Iyer, “Performance Characterization of TCP/IP Packet Processing in Commercial Workloads,” IEEE WWC-6, 2003.

[15] D. Marr et al., “Hyper-Threading Technology Architecture and Microarchitecture” Intel Technology Journal, 2002. http://www.intel.com/technology/itj/2002/volume06issue01/

[16] M. Martin, et al., “Token Coherence: A New Framework for Shared-Memory Multiprocessors,” IEEE Micro Special Issue, Nov-Dec 2003.

[17] N. Megido, “Adaptive Replacement Cache,” IBM T.J. Watson Research Center, http://www.almaden.ibm.com/cs/people/dmodha/arc-fast.pdf

[18] D. Minturn, et al., “Exploiting Architectural Techniques for Improving TCP/IP Processing Performance,” submitted to a conference.

[19] B. Nayfeh, K. Olukotun and J.P. Singh, “The Impact of Shared Cache Clustering in Small-Scale Shared Memory Multiprocessors,” Int’l Conference on High Performance Computer Architecture (HPCA-1), Feb 1996.

[20] J. B. Postel, “Transmission Control Protocol”, RFC 793, Information Sciences Institute, Sept. 1981.

[21] D.K. Poulsen and P.C. Yew, “Integrating Fine Grained Message Passing in Cache Coherent Shared Memory Multiprocessors,” Journal of Parallel and Distributed Computing, 1996.

[22] P. Ranganathan, et al., “The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems,” 24th International Symposium on Computer Architecture, June 1997, 144-156.

[23] A. Seznec, ``Decoupled Sectored Caches'', IEEE Transactions on Computers, Feb. 1997

[24] SimpleScalar LLC, http://www.simplescalar.com

[25] Y. Solihin, J. Lee, and Josep Torrellas. "Using a User-Level Memory Thread for Correlation Prefetching", The 29th Annual International Symposium on Computer Architecture (ISCA 2002), Anchorage, Alaska, May 2002.

[26] “SPECweb99 Design Document,” available at http://www.specbench.org/osg/web99/docs/whitepaper.html

[27] P. Stenstrom, “A Survey of Cache Coherence Protocols,” IEEE Computer, 1990.

[28] E. Suh, L. Rudolph and S. Devadas, “Dynamic Partitioning of Shared Cache Memory,” Journal of Supercomputing, July 2002.

[29] “The TTCP Benchmark”, http://ftp.arl.mil/~mike/ttcp.html

[30] D. M. Tullsen and S. J. Eggers. “Limitations of Cache Prefetching on a Bus-Based Multiprocessor,” Proc. 20th Annual Int. Symposium on Computer Architecture, pp.278-288, 1993.

[31] D.M. Tullsen, S.J. Eggers, and H.M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” 22nd International Symposium on Computer Architecture, 1995.

[32] VMware Inc., “VMware is Virtual Infrastructure”, http://www.vmware.com/vinfrastructure/

[33] C. A. Waldspurger, “Memory Resource Management in VMware ESX Server,” 5th Symposium on OSDI, 2002.

[34] W. A. Wulf and S. A. McKee. “Hitting the Memory Wall: Implications of the Obvious,” Computer Architecture News, 23(1):20--24, Mar 1995.

[35] L. Zhao, et al., “Efficient Cache Structures and Policies for Server Network Acceleration,” submitted to a conference.