

Exploring the Cache Design Space for Large Scale CMPs

Lisa Hsu[†], Ravi Iyer, Srihari Makineni, Steve Reinhardt[†], Donald Newell
Systems Technology Lab
Intel Corporation
{ravishankar.iyer, srihari.makineni, donald.newell}@intel.com

University of Michigan, Ann Arbor
[†]Advanced Computer Architecture Laboratory
{hsul, stever}@eecs.umich.edu

Abstract

With the advent of dual-core chips in the marketplace, small-scale CMP (chip multiprocessor) architectures are becoming commonplace. We expect a continuing trend of increasing the number of cores on a die to maximize the performance/power efficiency of a single chip. We believe an era of large-scale CMPs (LCMPs) with several tens to hundreds of cores is on the way, but as of now architects have little understanding of how best to build a cache hierarchy given such a large number of cores/threads to support. With this in mind, our initial goals are to prune the cache design space for LCMPs by characterizing basic server workload behavior in such an environment.

In this paper, we describe the range of methodologies that we are developing to overcome the challenges of exploring the cache design space for LCMP platforms. We then focus on employing a trace-driven approach to characterizing one key server workload (OLTP) in both a homogeneous and a heterogeneous workload environment. We study the effect of increasing threads (from 1 to 128) on a three-level cache hierarchy with emphasis on second and third level caches. We study the effect of varying sizes at these cache levels and show the effects of threads contending for cache space, the effects of prefetching instruction addresses, and the effects of inclusion. We make initial observations and conclusions about the factors on which LCMP cache hierarchy design decisions should be based and discuss future work.

1. Introduction

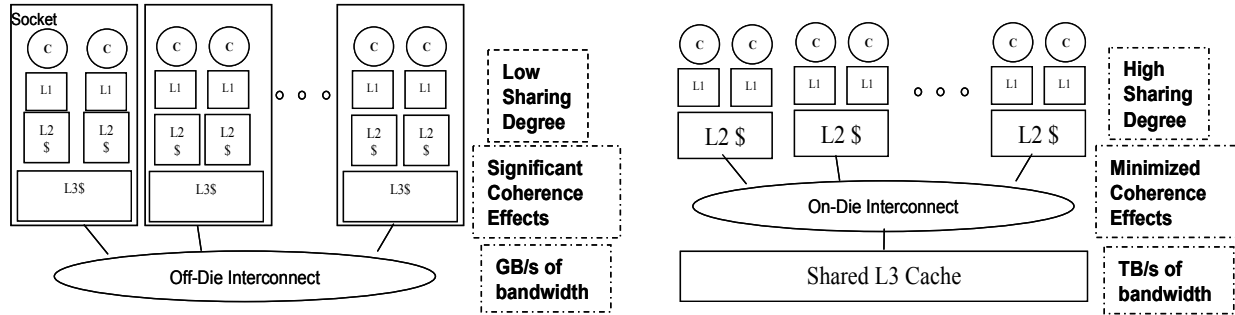
Small-scale chip multiprocessing (CMP) has arrived [4] and will soon become mainstream for most processor manufacturers. There are three key trends that have motivated the push for CMPs. (1) Relying on higher clock frequencies to deliver more performance has been slowly running out of steam as power/thermal constraints become increasingly restricting. (2) The

increasingly large gap between processor and memory speeds has made concurrent execution, whether in the form of threads or cores, critical for maintaining performance. (3) Advances in process technology are increasing the transistor budget to the point where the integration of more cores on the die is now feasible.

While CMP platforms are likely to proliferate across many application segments, server applications are likely to benefit the most since they have been shown to have high levels of thread-level parallelism (TLP) and thus can make good use of additional cores and threads. Some companies have already announced their plans of catering to throughput computing in the server market. The Azul Vega processor [1] will have 24 cores on a single die and is targeted to accelerating server workloads that run atop the J2EE platform. Sun Microsystems is increasing their throughput capacity by not only increasing the number of cores, but with multithreading [16, 10] as well, a combination they call CMT [14]. Their Niagara processor [7] has eight in-order cores, each capable of running four threads, for a total of 32 logical processors on a single chip. With manufacturers already using these two approaches to concurrency, we may very well see up to 128 logical processors on a single die within a decade.

For the remainder of this paper, we refer to processors capable of running a large number of threads simultaneously as LCMPs (large-scale CMPs). Additionally, we will use the term ‘thread’ to denote a logical processor, with no implications as to how many logical processors form a physical core. We do not study this aspect of designing an LCMP.

Instead, our focus is on exploring the cache hierarchy requirements of LCMP platforms. As we began our studies, we encountered significant challenges in the simulation of such platforms. Since LCMP is an emerging research area, there are very few, if any, established simulation infrastructures for exploring the cache design space. In this paper, we describe the set of tools that we are developing to address this void. We then describe our trace-driven cache simulation methodology that allows us to initially



(a) Multi-Socket Platforms with Few-Core Sockets

(b) Single-Socket Platforms with Numerous Cores

Figure 1. Today’s Architectures (Mostly Off-Die) versus Potential LCMP Architectures (Mostly On-Die)

prune the cache design space. We study both the thread/cache interplay of varying levels of threads and cache sizes, as well as some existing cache performance enhancing techniques in an LCMP environment. While such studies have been performed in the past, these studies have focused on non-CMP architectures, which exclude the possibility of large shared caches at the last level; or on CMP architectures on a significantly smaller scale [3]. We performed our studies for a key server workload (OLTP using TPC-C) in both a homogeneous and heterogeneous workload environment.

The rest of this paper is organized as follows. In Section 2, we introduce the large scale CMP platform and discuss its differences with current systems. In Section 3, we discuss simulation methodologies, both our own and overall, with respect to LCMP. Section 4 describes the comprehensive set of studies we have performed and our results. Section 5 concludes the paper with our observations and future work.

2. The LCMP Paradigm

Today’s server platforms (illustrated in Figure 1a) are largely based on multiple processor sockets, where each socket is a small-scale CMP (with a few cores on die). Our expectations are that tomorrow’s platforms will be LCMP platforms (illustrated in Figure 1b) with a significantly larger number of cores on die and potentially just one or two sockets in the platform.

2.1. Opportunities and Challenges

The two major design points in which LCMP architectures differ from today’s evolving architectures are in the feasible interconnect bandwidth and in the feasible cache hierarchy, which we discuss here:

Interconnect Bandwidth: As shown in Figure 1, today’s platforms have interconnects that are off-die and typically provide bandwidth on the order of several GB/s. By integrating multiple cores on-die, the interconnect bandwidth can conceivably reach TB/s.

This reduces both the logical and physical distance between the cores and the caches, which can significantly reduce the communication latencies between all threads on an LCMP architecture, as well as open the possibility for sharing a large last level cache (as in Figure 1b) while maintaining an acceptable last level cache latency. These allow for the exploration of new design points that are now free from the latency drawbacks they faced in multi-socket platforms.

Cache Sharing & Coherence: In today’s server processors, a significant portion of the die is occupied by the caches. Since multiple sockets have no direct way of sharing their caches, cache space can be under-utilized; meanwhile coherence effects reduce the overall cache efficiency by increasing the number of cache misses on each socket. As more cores are enabled on the die, the potential for sharing caches at multiple levels may address both these issues. Caches shared by multiple threads can increase cache utilization much the way execution cores shared by multiple threads via SMT can increase CPU utilization. Meanwhile, sharing can reduce the effects of coherence since data is updated in one central location. Coupled with an on-die interconnect, cache sharing may be beneficial.

On the flip side, the key challenge for LCMP caches is supporting the caching requirements of many threads in a single hierarchy. Threads must share the available cache space without unduly interfering with each other. In heterogeneous workload environments, it becomes especially important to enforce mechanisms to limit threads from trampling excessively on each other’s data. There has been work on this subject for small scale CMPs and SMT systems, but we believe the solutions may not be the same when the number of threads sharing a hierarchy are in the hundreds.

2.2. Design Considerations for an LCMP Hierarchy

Given a particular number of cores and total amount of cache space, there remain three types of design considerations. (1) “Vertical” design

considerations involve determining the optimal number of cache levels, and the amount of cache that should be doled out per level. (2) “Horizontal” design considerations involve the distribution of cache space within a cache level. For example, given n threads with X MB of cache space, should all n threads share all X MB? Or perhaps each thread should have X/n of private cache space? (3) Performance acceleration considerations are design techniques that have been used to accelerate cache performance in the past, but may need revisiting in the new LCMP paradigm.

To simplify our studies, we assume a baseline cache hierarchy like that which is pictured in Figure 1b, and thus we do not explicitly study vertical design considerations. We assume that this baseline cache configuration has three levels of caches—first level (FLC), mid-level (MLC) and last-level (LLC). We also assume the FLC level consists of per-core split caches with 32KB each for instructions and data, since for the moment we are only interested in the horizontal design aspects of the MLC and LLC. To this end, we study cache contention between varying numbers of threads at both these cache levels, as well as varying the amount of sharing between threads. We are also interested in several performance acceleration design points. Inclusion policies merit revisiting since there is a greater risk of wasteful redundancy when enforcing inclusion between many FLCs and an MLC, or between several MLCs and an LLC. We study code prefetching because one of the obvious benefits of sharing caches between threads in a server workload is that often different threads will run the same binary, and thus reduce code misses significantly. However, prefetching into non-shared caches may yield the same benefit without the risk of thread interference. We do not study the effects of basic cache design parameters, such as associativity (held constant at 16 ways for MLC and 32 ways for LLC) or line size (held constant at 64B for all caches). We also do not consider coherence policies and assume a typical MESI-based coherence protocol would be used.

3. Simulation Methodology and Tools

In this section we will discuss the challenges in evaluating an LCMP environment, present the set of tools and methodologies that we are developing, and finally describe the simulation infrastructure used for the studies we performed.

3.1. Simulating Hundreds of Threads – the Difficulties

One of the difficulties in simulating an LCMP cache hierarchy is the sheer number of threads that must

be simulated, which leads to challenges with both the speed of the simulation and the complexity of the simulation infrastructure. Another difficulty is quantifying the level of address sharing between many interacting threads. A more detailed discussion of each follows.

Many Threads: There has always been a tension between simulation speed and accuracy, even for uniprocessor architectural studies. With the research landscape opening up towards LCMP platforms, this tension will be further exacerbated by the number of threads being run simultaneously. If X million instructions of uniprocessor simulation gives an idea of how an application would perform in a certain uniprocessor system, $X * n$ million instructions may be required in an n -thread LCMP to achieve the same level of accuracy. This severe slowdown of execution speed for even the simplest of experiments limits the feasibility of adequately exploring the entire LCMP cache design space. Researchers are even now constantly looking for ways to reduce simulation times even without the LCMP paradigm, so the magnitude of this difficulty is significant.

Sharing and Interaction between Threads: Given the sharing discussion from Section 2.1, it is natural to explore possibilities of sharing caches between a few or even many threads. This is the “horizontal” design parameter discussed in the previous section. However, the sharing characteristics of highly threaded workloads must be well understood in order to accurately quantify the benefits of sharing caches between threads. Unfortunately, we are currently unable to gain this understanding using simulation because of the excessive simulation slowdowns on LCMP platforms.

The primary alternative to execution-driven simulation is trace-driven simulation. However, for a trace-driven simulation to reflect the nature of sharing between hundreds of threads, the traces either must be from a platform with hundreds of threads running a appropriately tuned workload, or the simulation must somehow inject sharing characteristics that accurately reflect sharing. It is a chicken and egg situation: in order to design good hardware we need to already have the hardware available to pull traces from it, or already fully understand it to model it accurately, in which case we would not need to model it any longer. Thus, this very exciting parameter for research of shared caches is difficult to evaluate because there is no way to know how and to what degree targeted workloads exhibit address sharing.

While there are other difficulties in LCMP cache design simulation, we feel these are the ones that will make the first order impact, and thus limit our discussion to these two.

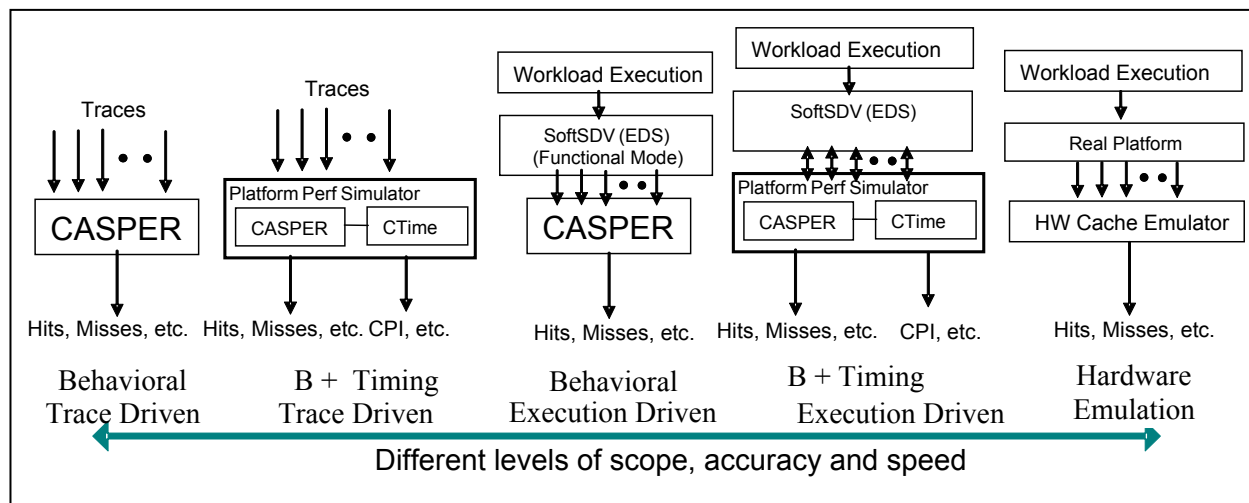


Figure 2. Simulation Methodologies and Tools (being developed) for LCMP exploration.

3.1. Simulating Hundreds of Threads – the Practicalities

Given the above difficulties, it is important to use a simulation methodology that adequately addresses both. Figure 2 shows a spectrum of methodologies that we are enabling for our research. As is to be expected, increasing accuracy leads to decreasing simulation speed, but we believe the aggregation of the entire suite will sufficiently address both difficulties discussed.

Most of our suite revolves around CASPER [6], a behavioral trace-driven cache simulator that provides great flexibility in simulating a wide variety of cache designs. For details, see the referenced paper; here we will focus on extensions for LCMP modeling.

As mentioned above, there are no workloads currently tuned for hundreds of threads, so we generated threads was by replicating trace files over and over again. Knowing that it is highly unlikely that all threads in a server workload would actually run the exact same code in lockstep, threads running from the same trace file were offset from each other so that they each had different starting points. Also, it is highly unlikely that hundreds of threads would touch the exact same addresses during execution, i.e. have 100% data sharing. To alleviate this realistically would require injecting a model for sharing into the simulation; however, as mentioned in the previous section, we do not have sufficient understanding of sharing to do this. For the time being, we have decided to use a rough model of sharing where 100% of code addresses are shared, and 0% of data addresses are shared. With respect to code sharing, we feel that this is not unreasonable, given that server applications often run the same binary on all threads. With respect to data sharing, however, this is just as unrealistic as the 100%

sharing scenario but with one key difference. Experiments with this type of sharing represent the worst case, and the behavior of a certain cache design can be bounded. We feel this bounding has great utility in the initial stages of research.

CASPER as a standalone tool represents purely behavioral trace simulation, the fastest form of simulation where the platform being simulated is fully parameterizable. The simulator function is just to model the behavior of a designated cache, given the memory stream(s) being fed from traces. This type of experimentation will yield cache miss information, which can be effective for providing boundaries and guiding the general direction of follow-up research. The data presented in this paper is based on this methodology.

We are also working on plugging a CASPER module into an Intel trace based platform simulator, where traces are fed into a timing simulator that offloads memory behavior modeling to CASPER. This simulation platform would provide performance data (in the form of cycles per instruction) as well as behavioral data, without being prohibitively slow. This simulation methodology can provide some validation as to whether the cache design guided by behavioral simulations would have performance improvements and not just lower miss rates. This simulation method will be slower than CASPER alone, but speed-wise is still feasible. With regard to sharing, since we are using this framework as a validation tool for the first set of experiments, we would use the same sharing paradigm.

Employing full-system execution driven simulation provides the greatest level of accuracy. As with trace based simulation, there is both behavioral and behavioral + performance modeling. This method is currently unfeasible for reasons previously mentioned; additionally, only some O/S'es can currently handle

booting on a platform with hundreds of threads, and as a result applications have not yet been tuned for this purpose. However, we have integrated CASPER with SoftSDV [17], a full-system simulator, with a goal of developing a system for LCMP simulations with large server workloads like OLTP (TPC-C).

Last, but not least, we are employing hardware emulation to perform a totally orthogonal function. Passive hardware emulation (much like in [9, 11]) allows an FPGA-based emulator to read code and data accesses that happen on a real platform in order to perform real-time cache simulation (programmed in the FPGA). We are collecting the performance of very large shared caches as well as some the sharing behavior of the workload (by monitoring bus traffic, we can see which addresses are being exchanged between which threads at the bus level). We hope this will provide insights into key characteristics like data sharing across threads that we can use to develop a sharing model.

3.3. Workloads and Configurations

We believe that server workloads are particularly suited to LCMP platforms because of the abundance of parallel threads. For this reason, our primary server workload is TPC-C [15]. We simulate TPC-C in both standalone mode as well as in the presence of other workloads (SPECjappserver and a synthetic streaming benchmark meant to represent iSCSI). The instruction traces (for TPC-C and SPECjappserver) that we used to feed into CASPER were collected from quad-processor machines. The memory reference streams are from a single processor within this quad-processor environment. To simulate the 0% data sharing mentioned in the previous section, data addresses were offset from their original values in the trace file such that each thread had its own separate address space. To avoid artificial set conflicts from doing this, we offset set values in each address as well. We have three and four traces for TPC-C and SPECjappserver, respectively. Instruction traces assigned to the (e.g. for

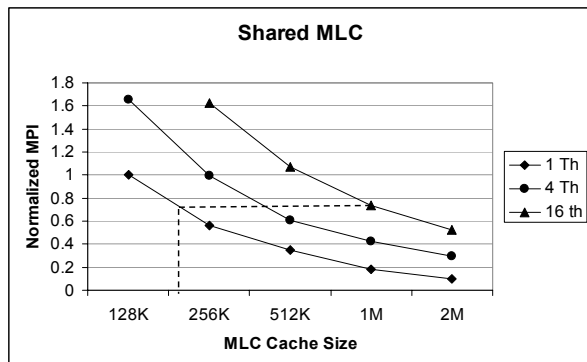


Figure 3. Impact of Thread Scaling on MLC Performance

16 threads and 4 traces, there will be 4 threads simulating each trace).

In Section 2, we described our baseline cache hierarchy. We are interested in how to design the MLC and LLC. The parameters of interest are (a) cache sizes, (b) threads sharing a cache, (c) inclusion policy, and (d) other performance acceleration techniques. We performed experiments with MLC cache sizes varying from a total of 128K to 2M, shared between 1 and 16 threads. On the LLC level, we varied cache sizes from 8M to 32M, shared between 16 to 128 threads.

4. LCMP Simulation Results and Analysis

In this section we present the data collected from LCMP cache simulations and analyze its implications on cache hierarchy design. All simulations were run for 20 million instructions per thread.

4.1. Thread Scaling Impact

We start by evaluating the effect of increasing the number of threads that share the MLC and the LLC in the LCMP architecture. Figure 3 shows MLC performance in terms of misses per instruction (MPI) as a function of cache size and the number of threads. The y-axis shows MPI normalized to the MPI of the single-threaded 128K case. There are two observations that can be made from this figure. The first one is that given a certain cache size, increases in threads sharing that cache size yield a disproportionately low increase in MPI (a good thing). For example, consider 1M data point, where the MPI has increased by roughly by less than a factor of 4 despite a 16-fold increase in the number of threads. This means sharing at the MLC level can scale quite well. The other observation can be seen by looking horizontally across the graph, as demonstrated by the dotted lines. The scenario where 1M is shared by 16 threads yields the same MPI as the scenario where 4 threads see roughly 420K of cache space and the scenario where each single thread sees

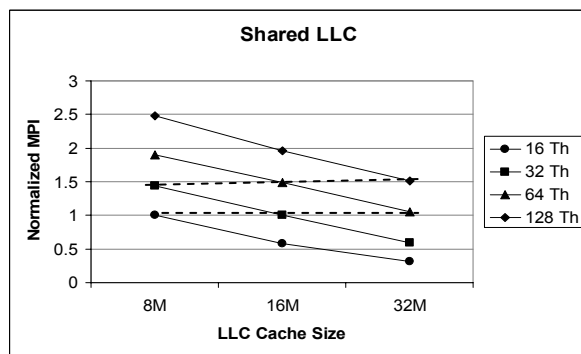


Figure 4. Impact of Thread Scaling on LLC Performance

about 240K of cache. In other words, 16 threads sharing a 1M cache results in 1.6X ($4 \times 420K/1M$) to 3.7x ($16 \times 240K/1M$) increase in cache efficiency as compared to the 1-thread and 4-thread scenarios. In the next subsection, we will discuss the breakdown of MPI for private and shared caches and show how such high cache efficiencies materialize.

Figure 4 shows the impact of thread scaling on the LLC. The x-axis shows the cache size ranging from 8M to 32M and the curves in the figure are for varying numbers of threads (16 to 128). The LLC performance scaling is very different from the MLC performance scaling. Increasing the number of threads from 16 to 128 (8X factor) increases the MPI by 3.5X for a 16M LLC. This still demonstrates some sharing benefit, but not as strongly as the MLC case. This is because the LLC performance is less sensitive to code sharing since it is mostly comprised of data and not code. Even so, it should be noted that the MPI does not increase too drastically as more threads are added. Specifically, the dotted line in the figure shows that doubling the number of threads and doubling the cache size results in a very minor increase in MPI. This shows that for homogeneous workloads that employ many threads, a moderately-sized but shared LLC may perform reasonably well. We will present the same scenario for heterogeneous workloads in a later section.

It is important to remember that these numbers represent 0% data sharing. Doing these studies with a data sharing model would likely yield even better results, since total working set would probably be reduced.

4.2. Private vs. Shared Caches

The purpose of this study is to evaluate and understand the degree of cache sharing that is minimally desirable at each cache level. For example, using the LCMP platform architecture shown in Figure 1b as reference, we would like to evaluate whether it is more beneficial to share a larger cache between large numbers of threads or to have smaller subsets of threads share smaller caches.

Figure 5 shows the overall MPI comparison between a private versus shared cache at the MLC. We assume 16 threads. By private, we mean every 4 threads get a private cache, while for shared, we mean all 16 threads share the entire MLC level. The overall cache space is kept constant during this comparison and is shown in the x-axis. The graph shows that the difference in normalized MPI between private and shared caches is significant at 512K (~35% reduction), somewhat less significant at 1M (~25% reduction) and is low at 2M (~6%). The key observation is that for smaller MLC sizes, sharing can bring noticeable benefit

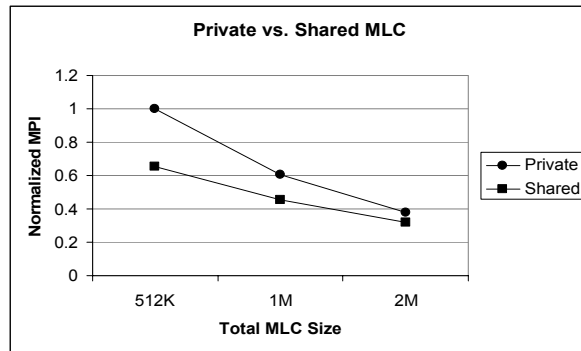


Figure 5. Performance of Private vs. Shared MLC

even when just sharing code lines. For larger sizes, there is less benefit since the code working set size begins to fit in even the private caches. However, it is reasonable to infer that if there were more significant cache sharing, the benefit would likely further increase. The good news is that despite completely separate data addresses spaces, shared caches still manage to glean benefit from sharing only a small percentage of total address requests.

The benefit that we saw with shared caches over the private caches was largely due to the existence of a high degree of code sharing. We wanted to determine whether another code MPI reducer, prefetching, would impact the benefits we observed. We implemented code prefetching in the CASPER to prefetch “n” subsequent cache lines upon a miss in the mid-level cache. Figure 6 shows the effect of code prefetching on shared and private caches. The normalized MPI (broken down into data and code MPI) is shown as a function of private vs shared configurations for various cache sizes and for “n” (0, 2, 4). Several observations can be made from the data: (1) Code prefetching benefits private caches more than it does shared caches, (2) However, even after code prefetching, the private cache MPI remains moderately higher than that of the shared cache. (3) Prefetching two lines of code is better than prefetching four lines of code since the latter causes enough pollution to increase the overall MPI. One caveat is that the effect of prefetching is very timing dependent. Since we do not

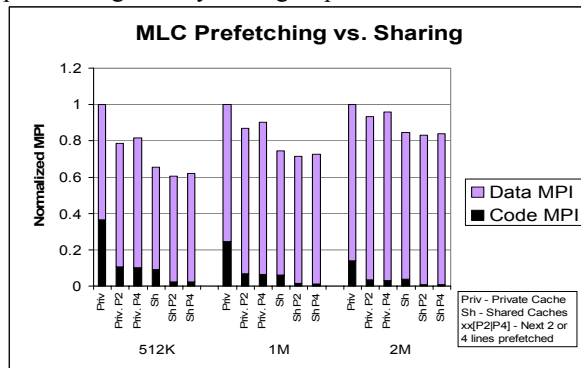


Figure 6. Effect of code prefetching on MLC

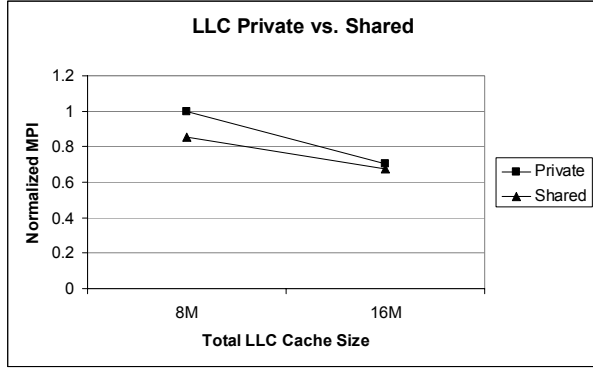


Figure 7. Performance of Private vs. Shared LLC.

have timing information in the simulator, it is hard to determine what the exact effect of prefetching is on these caches. We plan to evaluate this further in a platform performance simulator.

Figure 7 shows the results of our private versus shared study in an LLC context. The overall LLC size is varied from 8M to 16M – in the shared scenario, there are 128 threads sharing the overall cache space; whereas in the private configuration, there are 16 threads each sharing a 1M to 2M cache. The performance at this level is dominated by data MPI, and despite 128 distinct datasets using one cache, there is not enough interference between them to reduce performance below that of private caches.

4.3. Impact of Multiple Levels and the Inclusion Policy

All of the data presented until now was collected by simulating a single-level of cache sized appropriately for the level. In this subsection, we evaluate the effect of enabling multiple levels of cache. A key design aspect here is to determine whether the caches should be inclusive or non-inclusive (i.e. not enforcing inclusion, as opposed to exclusion). Figures 8 and 9 show the normalized MPI for the MLC and LLC respectively. The x-axis shows the cache sizes and the

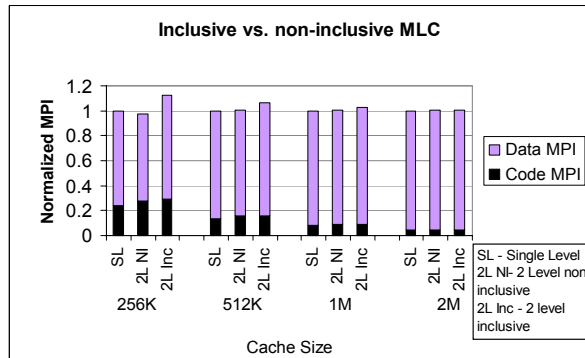


Figure 8. Comparison of Inclusive vs. Non-Inclusive MLC

inclusion policy or the use of single-level simulations.

In the MLC case (Figure 8), we assume 256K of FLC total. The data indicates that at larger cache sizes and 2M), there is not much of a difference between single and multi-level caches nor between inclusive and non-inclusive caches. However, at smaller cache sizes (256K, 512K), non-inclusive caches outperform both the single level cache as well as the non-inclusive cache by a slight margin. This is because of excessive redundancy. One more subtle but interesting observation is that multi-level caches (inclusive and non-inclusive) have higher code MPI than the single level because there is a higher probability that the code gets evicted from the larger level since the frequency of accesses gets hidden by the smaller cache.

In the LLC case, we assumed 4MB total of MLC. The data (Figure 9) again shows there is no difference between these three configurations at the largest (32M) cache size. At 8M and 16M sizes, non-inclusive cache performed slightly better than the inclusive cache. The discussion about inclusive vs. non-inclusive would not be complete without talking about amount of snoop traffic that is generated in each configuration and the bandwidth requirements on the caches. We intend to study this aspect in the near future. Regardless, we can see that it may be worth our while to have non-inclusive caches when the total size of the previous cache level is a relatively significant fraction of the next level, since redundancy costs are greater.

4.4. Heterogeneous Workloads

In this subsection, we try to understand how emerging usage models impact cache performance. iSCSI is fast becoming a common way to connect to storage systems. Since iSCSI runs on top of the TCP/IP protocol stack, it introduces lot of streaming data [19] during packet processing – this has the potential to pollute large portions of the cache. We wanted to understand the impact of introducing streaming data on the same platform running OLTP and assess its impact

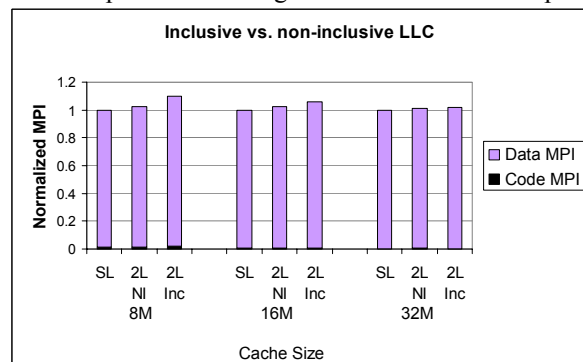


Figure 9. Comparison of Inclusive vs. Non-Inclusive LLC

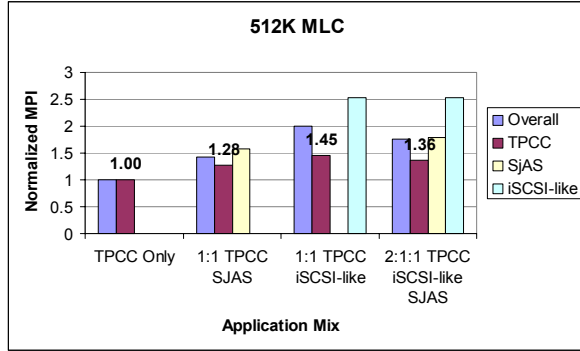


Figure 10. TPC-C MLC MPI in Heterogeneous Scenarios

on the misses faced by the OLTP application. Another trend, platform virtualization [2, 18], leads to the possibility of running multiple disparate applications on the same machine. The LCMP platform could accelerate the deployment of these new usage models, we evaluated their performance. Our main motivation here is to understand the effect of simultaneous execution of these different workloads while sharing the cache as a common resource that typically does not have any quality of service mechanisms [5] built into it.

Figures 10 and 11 show data from simulations running a mix of TPC-C, iSCSI-like streams, and SPECjappserver (SjAS [12]). The data is normalized to the scenario where only TPC-C is running on all the threads. Figure 10 shows the normalized MPI data for a 512K MLC. The ratios shown in the x-axis indicate how many threads are (relatively) allocated to each workload. For example, “2:1:1” implies that out of 16 threads, 8 are dedicated to TPC-C, 4 are dedicated to iSCSI-like streams and 4 are dedicated to SjAS. The graph clearly shows how TPC-C MLC MPI increases when mixed with other workloads. Adding a streaming workload that runs simultaneously with TPC-C has the most significant impact on TPC-C MPI (~45% increase).

Figure 11 shows similar data, but for the LLC. Here we have simulated 128 threads on a 16M LLC. Even for the LLC, we see the most significant impact caused by the streaming workload (2.14X increase in TPC-C MPI). The impact is much more pronounced on LLC than it is on MLC. We expect that incorporating hardware mechanisms for enforcing QoS in the cache hierarchy to handle heterogeneous scenarios is crucial to the performance of LCMP architectures.

4.5. On-die and Off-die Bandwidth Analysis

One major consideration when designing the LCMP cache hierarchy is to understand the bandwidth requirements that will be imposed on the on-die interconnect and off-die memory subsystem. For

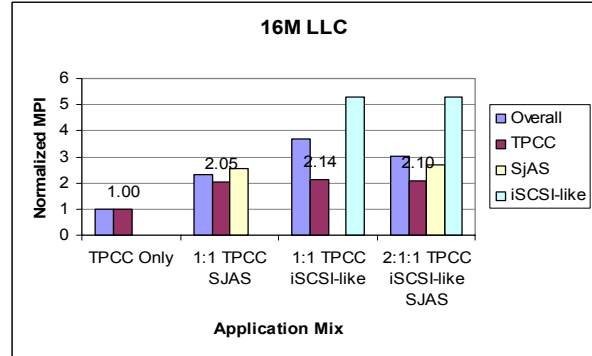


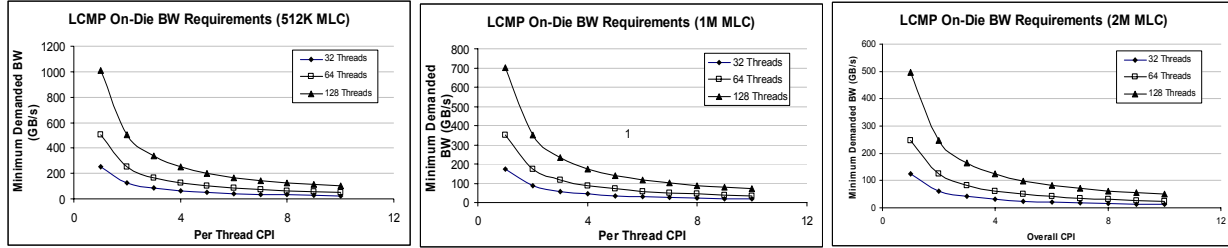
Figure 11. TPC-C LLC MPI in Heterogeneous Scenarios

example, the LCMP architecture illustrated in Figure 1b clearly shows that the on-die interconnect and last-level cache bandwidth requirements will be based on MLC miss rates. Similarly, memory bandwidth requirements will be based primarily on the LLC miss rate. In this section, we estimate the bandwidth requirements for the several of the cache hierarchy options that we evaluated. It should be noted that these bandwidth requirements are the lower bound since they are based solely on the demand misses and do not take into account I/O traffic, snoop traffic, and similar other traffic that will also be introduced into the subsystems of interest.

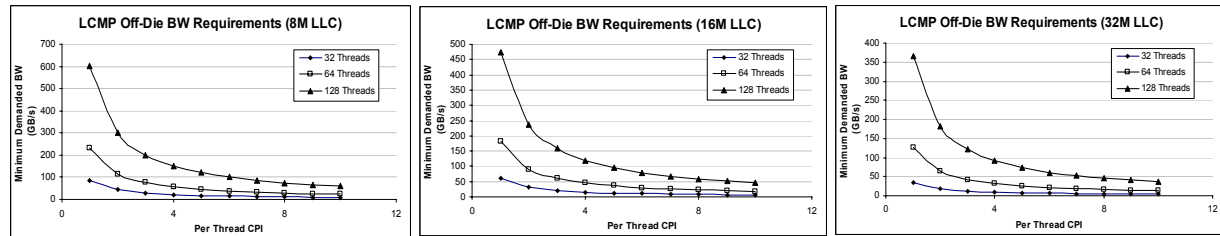
Figure 12 shows the on-die and off-die bandwidth requirements for the interconnect and LLC (Figure 12a) and for the memory subsystem (Figure 12b). The bandwidth requirements are a function of the misses per instruction, the cycles per instruction (per thread) and the number of threads simultaneously executing. In the figures, we vary the CPI from 1 to 10 (as a sensitivity study) and show the requirements for 32 to 128 threads.

Figure 12a shows three figures for on-die bandwidth requirements of 512K, 1M, and 2M mid-level caches (where each mid-level cache is shared by 16 threads), while Figure 12b shows the same for the LLC at 8, 16, and 32M. As an example, the demanded bandwidth (at a CPI of 4) ranges from ~250 GB/s for 512K caches to ~125 GB/s for 2M mid-level caches, and ~150 GB/s for an 8M LLC to 90 GB/s for a 32M LLC. (Note that actual system design will require even higher bandwidths to cover peak conditions.) For obvious reasons, bandwidth demands across all the graphs increase when core compute power is less; additionally more threads mean higher demands, also an intuitive result.

This analysis was done in isolation from other aspects of total cache system design. It should be noted however, that when the time comes to actually design an entire LCMP cache hierarchy, tradeoffs between interconnect and cache design need to be considered holistically, as shown by Kumar [8].



(a) On-die bandwidth requirements for TPC-C in LCMP platforms



(b) Off-die bandwidth requirements for TPC-C in LCMP platforms

Figure 12. On-Die and Off-Die Bandwidth Demands (Minimum) as a function of core performance

5. Conclusions and Future Work

In this paper, we motivated the need for exploring the cache design space for large-scale CMP platforms. We showed that large-scale CMP platforms (as compared to traditional multi-socket platforms) introduce some opportunities like high available on-die bandwidth and the possibility of shared caches at multiple levels in the hierarchy.

One of the key challenges in exploring the cache design space is in enabling the right set of methodologies and tools to yield meaningful results. We described the challenges on this front, as well as what we are doing to meet these challenges. Finally, we provided an overview of the trace-driven simulation methodology that we used for the first phase of exploration covered in this paper.

Our main contribution is the analysis of the impact of increasing threads on LCMP cache performance for TPC-C. Using trace-driven simulation, we made a number of key observations:

- (1) Sharing mid-level caches between 16 threads can provide a 1.6X to 3.7X space efficiency benefit for TPC-C.
- (2) Code prefetching improves the performance caches significantly, but is not sufficient enough to close the performance gap with the shared caches.
- (3) Even with no data sharing simulated, a 32-way last-level cache scales reasonably well when contended for by up to 128 threads.
- (4) Inclusion does not seem to have significant impact on cache performance as long as the difference in cache size between the levels is larger than 2X.

(5) A mix of heterogeneous workloads affects the TPC-C cache performance significantly. QoS mechanisms that enforce fairness and prioritization are needed to protect threads from each other when multitasking.

(6) On die and off-die bandwidth demands will play a significant role in determining the optimal cache hierarchy for LCMP platforms.

The studies presented in this paper should be viewed as preliminary. We plan to expand our workload mix (similar studies for workloads like SPECjappserver [11] and SAP [12] are already underway), and we intend to deepen the detail and rigor of our studies as we continue down the path pruned by these initial results. We especially believe that the heterogeneous scenario needs to be investigated in more detail in order to determine the appropriate QoS mechanisms necessary.

NOTICES

® is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

REFERENCES

- [1] “Azul Compute Appliance,” Azul Systems, can be found at http://www.azulsystems.com/products/epools_cappliance.html

- [2] P. Barham, B. Dragovic, et al., "Xen and the art of virtualization," In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003), NY, USA, Oct 2003.
- [3] L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," IEEE Computer, 30(9), 79 - 85, September 1997. also see <http://www-hydra.stanford.edu/>
- [4] Intel Corporation. "Intel Dual-Core Processors -- The First in the Multi-core Revolution," <http://www.intel.com/technology/computing/dual-core/>
- [5] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," 18th Annual International Conference on Supercomputing (ICS'04), July 2004.
- [6] R. Iyer, "On Modeling and Analyzing Cache Hierarchies using CASPER", 11th IEEE/ACM Symposium on Modeling, Analysis and Simulation of Computer and Telecom Systems, Oct 2003.
- [7] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 25, 21-29, Mar. 2005
- [8] R. Kumar, V. Zyuban, and D. Tullsen, "Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling", 32nd International Symposium on Computer Architecture, June 2005
- [9] S-L. Lu and K. Lai, "Implementation of HW\$im - A Real-Time Configurable Cache Simulator," FPL 2003: 638-647
- [10] D. Marr, F. Binns, et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, Vol 3, Issue 1, Feb 2002, can be found at ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf
- [11] A. Nanda, K. Mak, K. Sugavanam, R. K. Sahoo, V. Soundararajan, and T. Smith, "MemorIES: A programmable, real-time hardware emulation tool for multiprocessor server design," ACM SIGPLAN Notices, Vol. 35, Issue. 11, Nov. 2000.
- [12] SPECjAppserver2004 User's Guide, <http://www.spec.org/jAppServer2004/docs/UserGuide.html>
- [13] Sap America Inc., "SAP Standard Benchmarks," <http://www.sap.com/solutions/benchmark/index.epx>
- [14] L. Spracklen and S. Abraham, "Chip Multithreading: Opportunities and Challenges," Industrial Session, 11th International Conference on High Performance Computer Architecture (HPCA-11), San Francisco, 2005.
- [15] "TPC-C Design Document", available online on the TPC website at www.tpc.org/tpcc/
- [16] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism," in 22nd Annual International Symposium on Computer Architecture, June 1995.
- [17] R. Uhlig, R. Fishtein, et. al., "SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture. Intel Technology Journal. Q4, 1999. (<http://www.intel.com/technology/itjf>)
- [18] R. Uhlig, G. Neiger, D. Rodgers, et al., "Intel Virtualization Technology," IEEE Computer Vol 38, Issue 5, May 2005.
- [19] L. Zhao, R. Illikkal, S. Makineni and L. Bhuyan, "TCP/IP Cache Characterization in Commercial Server Workloads," 7th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-7), held along with HPCA-10, Feb. 2004.