CMPSched$im: Evaluating OS/CMP Interaction On Shared Cache Management

Jaideep Moses, Konstantinos Aisopos (Princeton University), Aamer Jaleel, Ravi Iyer,
Ramesh Illikkal, Don Newell, Srihari Makineni

Hardware Architecture Laboratory, Intel Corporation
Contact: jaideep.moses@intel.com

ABSTRACT

CMPs have now become mainstream and are growing in complexity with more cores, several shared resources (cache, memory, etc) and the potential for additional heterogeneous elements. In order to manage these resources, it is becoming critical to optimize the interaction between the execution environment (operating systems, virtual machine monitors, etc) and the CMP platform. Performance analysis of such OS and CMP interactions is challenging because it requires long running full-system execution-driven simulations. In this paper, we explore an alternative approach (CMPSched$im) to evaluate the interaction of OS and CMP architectures. In particular, CMPSched$im is focused on evaluating techniques to address the shared cache management problem through better interaction between CMP hardware and operating system scheduling. CMPSched$im enables fast and flexible exploration of this interaction by combining the benefits of (a) binary instrumentation tools (Pin), (b) user-level scheduling tools (Linsched) and (c) simple core/cache simulators. In this paper, we describe CMPSched$im in detail and present case studies showing how CMPSched$im can be used to optimize OS scheduling by taking advantage of novel shared cache monitoring capabilities in the hardware. We also describe OS scheduling heuristics to improve overall system performance through resource monitoring and application classification to achieve near optimal scheduling that minimizes the effects of contention in the shared cache of a CMP platform.

1. INTRODUCTION

In this era of chip-multiprocessor (CMP) platforms [1, 4, 5], performance improvement is being provided in every generation by increasing the number of cores integrated on-die. Apart from cores, a significant amount of die space is also devoted to providing large amounts of shared resources such as last-level cache. For example, the latest quad-core Intel Xeon Processor 5400 series [4] consists of four identical cores, with each pair of cores sharing a 6MB last-level cache. In addition, several CPU manufacturers are considering integrating additional heterogeneous elements on-die for special-purpose computing. As CMP processors continue to evolve, the management of resources on these processors requires careful interaction between the Operating System (OS) and the CPU architecture. In this paper, we focus on the interaction between the OS and the CMP architecture to identify better approaches for performance analysis.

Performance analysis of OS and CMP interaction is challenging because it typically requires full-system execution-driven simulation. It is well understood that full-system execution-driven simulators are not only time-consuming to develop but are also time-consuming to execute. As a result, it is highly desirable to find alternative approaches for fast and flexible exploration of OS/CMP interactions with reasonable accuracy. In this paper, we present CMPSched$im, a simulation infrastructure that achieves this goal with a specific focus on shared resource management.

CMPSched$im uses a binary instrumentation tool (Pin [14]), a multi-programmed cache simulator (CMP$im [15]) and a Linux scheduler simulator (Linsched [16]) all integrated to form a highly robust and flexible simulation infra-structure. CMPSched$im attempts to overcome the limitations of existing trace based and execution-driven tools by offering the flexibility and ease of use of trace driven simulation while accurately capturing the interactions between the simulated hardware and the OS/execution environment. We describe how we created CMPSched$im by integrating CMP$im and Linsched and then present a detailed case study on the use of CMPSched$im to study the interaction of OS and CMP on shared resource management. The study shows OS scheduling heuristics can be designed to employ future shared cache monitoring counters to improve overall system performance through resource aware scheduling based on proposed cache occupancy monitoring as well as existing hardware counters in the presence of multiple workloads.

The rest of the paper is organized as follows. Section 2 motivates the problem of contention in the last-level shared cache and the need for accurately capturing interactions between the OS/execution environment and hardware architecture. We also discuss related work on performance analysis tools for OS/CMP interaction in Section 2. We present our proposed evaluation framework (CMPSched$im) in Section 3 and show how the gap between trace driven and execution driven simulations can be bridged. Section 4 describes a case study showing the use of CMPSched$im to model a set of workloads running on a CMP system and the OS scheduler employing shared cache monitoring (existing and proposed counters) to improve overall throughput. The results show the benefits of employing cache misses and shared cache occupancy individually as well as together

978-1-4244-4184-6/09/$25.00 ©2009 IEEE

when scheduling workloads on a CMP platform. In Section 5 we present our conclusions and future work

2. BACKGROUND AND RELATED WORK

Effective interaction between the OS/CMP architecture is critical to optimize the management of shared platform resources. Future workload scenarios will consist of heterogeneous workloads running simultaneously to take advantage of the abundant hardware thread parallelism provided in the platform. With these scenarios emerges the problem of contention in the shared cache among the heterogeneous workloads. Recent studies in [2, 6, 7, 8, 11, 12] indicated that contention in the shared cache causes significant loss in performance and Quality of Service (QoS [6]). We ran SPEC CPU2000 applications on the Core 2 Duo desktop (2 cores sharing 4MB of last-level cache) in two modes: (a) dedicated mode where we run one application on one core in the platform and (b) pair-wise mode where we run the preferred application on one of the cores and run another application on the other core. Figure 1 shows the slowdown in terms of the ratio of execution time in pair-wise mode over that in dedicated mode. As one can observe, the slowdown can be as high as ~4.76X (when mcf is running with swim). Overall, we observe that in 30% of the pairs, the slowdown ranges from 1.2X to 5X. We have performed similar studies with virtualized server workloads running simultaneously and observed similar slowdown effects.

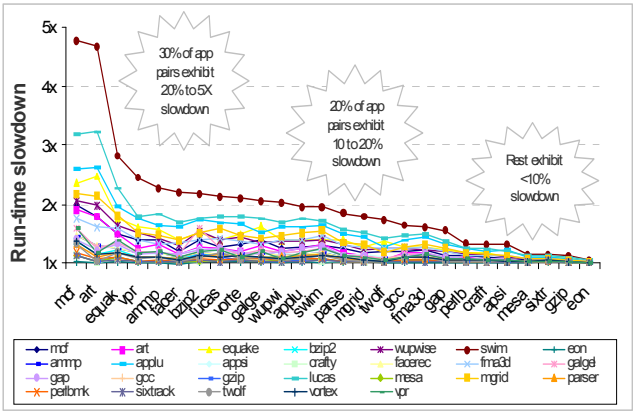

Figure 1. Impact of Shared Cache/Memory contention

As the number of cores and on-die shared resources increase, managing the shared resources requires efficient interaction between the operating system and the CMP platform. For example, QoS policies [6, 7, 12] have been proposed to guide the partitioning of shared cache and thereby address the issue of contention in the shared cache and memory. By monitoring resource usage in the shared cache, an OS or VMM can make more intelligent scheduling decisions by deciding which applications can run together to minimize contention in the shared cache. Previous studies [3, 7, 13] have alluded towards how such scheduling decisions can be made by monitoring capacity or interference among applications. However these studies are based on trace-driven simulations which do not accurately capture the interactions between the OS and the CMP platform. Even when the interactions between the OS and CMP platform architecture have been captured, a comprehensive study on how applications can be classified through resource monitoring to achieve near optimal scheduling is an extremely challenging problem especially in real world scenarios where there are more applications than available hardware threads.

Studying the interactions between the OS and the underlying CMP architecture requires simulators that (a) model the underlying CMP architecture (b) can concurrently execute arbitrary number of applications (c) model an OS scheduler to schedule the concurrently executing applications on the CMP platform. Such features are commonly found in full-system execution driven simulators which have the drawbacks of being extremely slow. Alternatively, trace-driven simulators, which are generally much faster than full-system execution driven simulators, have been used. However, trace-driven simulators have the drawbacks of dealing with large address trace files that potentially already have the native OS scheduling policies captured during the trace collection process. Additionally, conventional trace-driven simulators are single-threaded and can become extremely slow with large number of applications and cores on the CMP. Recently, CMP$im [15], a Pin-based on-the-fly multi-threaded performance simulator was proposed. CMP$im avoids the use of large address trace files by extracting memory addresses from an application on-the-fly using binary instrumentation. CMP$im is also a fast parallel simulator that models separate software threads for each application executing on the simulated CMP. CMP$im, however, lacks an OS scheduler and is thus unable to model real world scenarios where the number of concurrently executing applications is greater than the total number of cores. To model such scenarios, we enhance CMP$im with Linsched [16], a recent simulator that models the Linux scheduling behavior. By combining the benefits of Pin, CMP$im and Linsched, we created CMPSched$im, a simulation platform that overcomes the drawbacks of existing execution driven and trace driven simulators to study the interactions between the OS/CMP platform

3. CMPSCHED$IM: A FAST AND FLEXIBLE FRAMEWORK FOR OS/CMP INTERACTION

In this section, we introduce the CMPSched$im evaluation framework and describe details on how it was developed. We start below by describing CMP$im, a tool that we started with and extended to realize CMPSched$im. The primary goal is to capture the interactions between the OS scheduler, CMP architecture and the workload and simulate it at a high speed. To undergo this effort, we utilize CMP$im [15], a Pin [14] based parallel multi-core performance simulator. To study the interactions between the OS and CMP platforms, we created the CMPSched$im simulation framework by enhancing CMP$im with a Linux scheduler Linsched [16]. We describe CMP$im, Linsched, and the required enhancements to CMP$im to support the OS scheduler. The

baseline CMP$im runs at speeds in the order of 10s of MIPS for single-threaded workloads and 4-5MIPS for multi-programmed workloads where the number of applications do not exceed the number of simulated hardware threads. CMPSched$im also achieves similar speeds.

CMP$im, is a memory system simulator that utilizes the Pin binary instrumentation system to evaluate the performance of single-threaded, multi-threaded, and multi-programmed workloads on a simulated single or multi-core processor. CMP$im models a simple processor pipeline model and uses Pin as the functional model to dynamically feed instructions and memory addresses (from workloads under study) to the simulated cores. In doing so, CMP$im conducts on-the-fly multi-core studies of concurrently executing workloads and avoids the I/O overheads associated with large address trace files.

To simulate the behavior of multiple programs executing simultaneously on a simulated CMP, CMP$im makes use of shared memory. This was necessary because user-level binary instrumentation tools, such as Pin, can only instrument a single application at a time. The multi-programmed CMP$im model creates the simulated processor model in shared memory and requires multiple instances of Pin and CMP$im to attach themselves to the shared memory (as illustrated in Figure 2). Due to the absence of an application scheduler in the baseline CMP$im model, CMP$im requires that the number of concurrently running applications be at most the number of cores (or hardware-threads) on the simulated CMP. Simulation starts once the applications register themselves with the processor model. CMP$im models timing by periodically synchronizing the cycle counts on each simulated core using barrier synchronization.

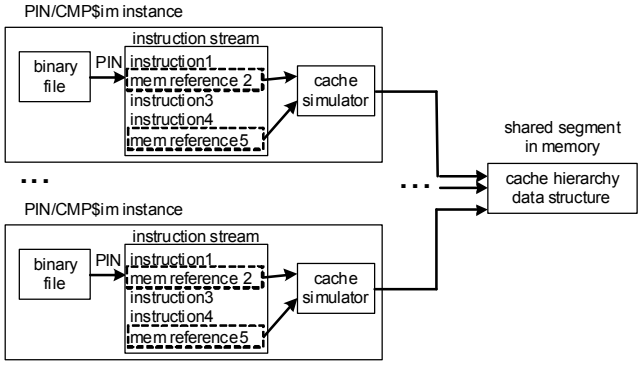

Figure 2. Overview of Multi-Programmed CMP$im

In order to support more applications than simulated hardware threads, CMP$im had to be improved to incorporate three major components: 1) a blocking mechanism to support multiple applications per hardware thread, to ensure that no two applications can run on the same hardware thread at the same time, 2) an OS scheduler to handle the scheduling of applications to hardware threads and determine how much time they run on each hardware thread and 3) modifying the existing barrier synchronization in CMP$im. In addition to the above we needed 4) a mechanism to support application migrations from core to core, to guide the OS scheduler to make more intelligent scheduling decisions based on 5) resource usage monitoring counters. A block diagram of the enhanced CMPSched$im is illustrated in Figure 3.

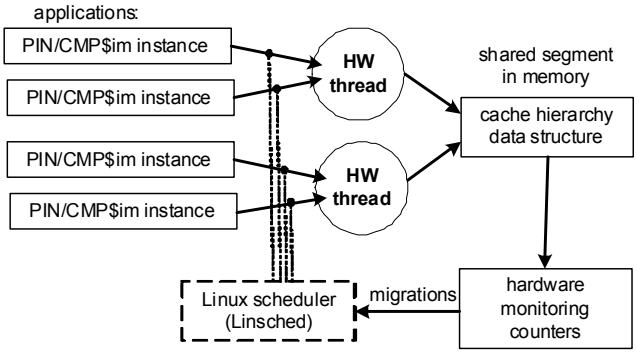

Figure 3. Overview of CMPSched$im

3.1. Support for Multiple Applications Per Hardware Thread

Enabling multiple applications per hardware thread requires implementing a blocking mechanism, which guarantees that no more than one application can run in each hardware-thread at any time. This was achieved by introducing a per-core lock. The array of per-core locks was defined at the shared segment, in order to be visible to all applications. Possessing a core lock implies that the application is running on the corresponding hardware thread. The application releases the lock once its time quota has expired, or when it is terminated.

3.2. Integration of OS Scheduler Simulator

A Linux scheduler simulator, Linsched [16], was integrated to capture real-system application scheduling behavior. Linsched is an independent module that takes as input the number of cores and application IDs, and makes global decisions about which application will be scheduled in which core and when. Linsched uses the latest Completely Fair Scheduler (CFS) kernel at its core, to accurately simulate scheduler behavior. Using APIs into Linsched, all applications store locally a schedule, which indicates whether they should run/block at any given time. This schedule is updated at regular time intervals by requesting the scheduling decisions for this application, for a certain time period, from Linsched.

An example is shown in Fig.4: An instance of Pin/CMP$im that represents the application with appID=2 has as input the global clock variable, which is a shared variable among all Pin/CMP$im instances, and indicates the global time for all applications. In addition, there's a local schedule table indicating when this application may run. Once the global clock exceeds the maximum timestamp of this table, the application needs to request the schedule for the next scheduling period from the Linsched module. In the current example, global clock is 7ms and has exceeded 6ms, which was the latest update for the schedule table. Consequently, a request to get the schedule for the next 6ms is sent to Linsched. Linsched will filter the entries that

correspond to appID=2 and fall into the requested time period (i.e. [6ms, 9ms]) and respond to the Pin/CMP$im instance. The Pin/CMP$im simulation may only run if there's a record at the local scheduling table, indicating that the application is currently scheduled to some core. It is also notable that since Linsched will never schedule more than one application to a core at any specific time, there is no contention for the core lock. Typically, only one application will try to acquire it, right after the application that was previously scheduled to this core will exceed its time quota and release it.

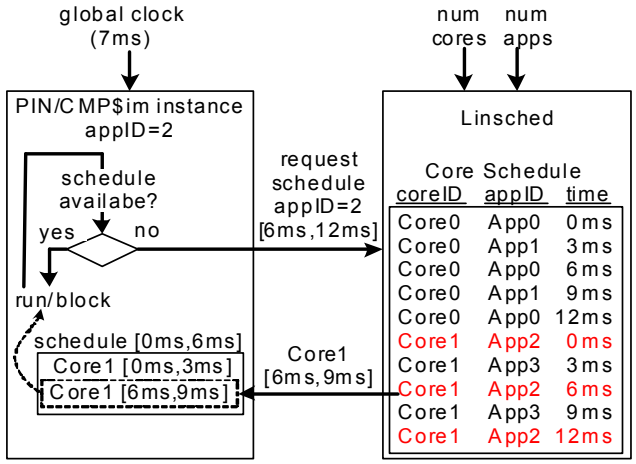

Figure 4. Communication between Pin/CMP$im module and Linux Scheduler (Linsched)

3.3. Modifying CMP$im Synchronization Barrier

CMP$im models timing by periodically synchronizing the cycle counts of each application using barrier synchronization. When the number of applications exceed the number of cores available (as in CMPSched$im), synchronizing the cycle counts of all applications hinders forward progress for running applications because they would be waiting for blocked/waiting applications to make progress. To ensure forward progress we (a) only synchronize the running applications (b) "catch up" newly scheduled applications by providing them with an indication of the current global time.

To illustrate this, Figure.5, shows four applications concurrently executing on a simulated 2-core CMP. The scheduler starts off by scheduling applications 0 and 2 to run while applications 1 and 3 are waiting. Let us assume that running applications synchronize with each other every 1000 simulated cycles. Once one of the running applications, say 0, reaches the first barrier, it has to wait for the rest of the running applications to reach the barrier (i.e. application 2). Note that since only running applications wait in the barrier, the number of applications that will synchronize in every barrier boundary is equal to the number of cores. Once application 2 finishes simulating a 1000 cycles, the barrier is released and moved to 2000. Consequently, both applications 0 and 2 resume execution. Now, suppose a scheduling update causes application 3 to be scheduled to core 1 (application 2 is blocked). Application 3 is "caught up" by updating its cycle count to match the previous barrier boundary (i.e. 1000), so as to reflect the progress of the global clock while application 3 was waiting to execute.

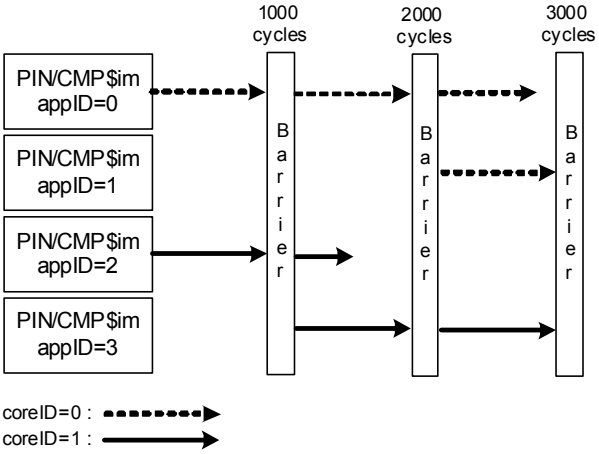

Figure 5. Example illustrating the usage of the new synchronization barrier

3.4. Support for Application Migration

CMPSched$im supports pinning (affinitizing) applications to cores at run time. Pinning an application to a core can be achieved by a function call to Linsched, which forces Linsched to migrate the application to the requested core in the next scheduling period. Whenever an application exceeds its time quota and releases its core lock, it checks whether the core schedule requires that the application run on a different core in the next time quota. If this is the case, the application will compete for the other core when attempting to switch back in possibly resulting in cold start misses in the memory hierarchy of the newer core.

3.5. Monitoring Counters which Measure per Application Capacity and per-Appliction MPI

Monitoring counters that monitor per application capacity and per application Misses per Instruction (MPI) are supported in the enhanced simulation infrastructure. A heuristic, which gets as input the values of these hardware monitoring counters (Figure 6), decides on the optimal mapping of applications to hardware threads, so as to improve overall system performance. When Linux Scheduler (Linsched) is generating a scheduling table indicating when each application will run, which takes place at the end of each scheduling period; it has to comply with the application-to-core mapping request of the heuristic.

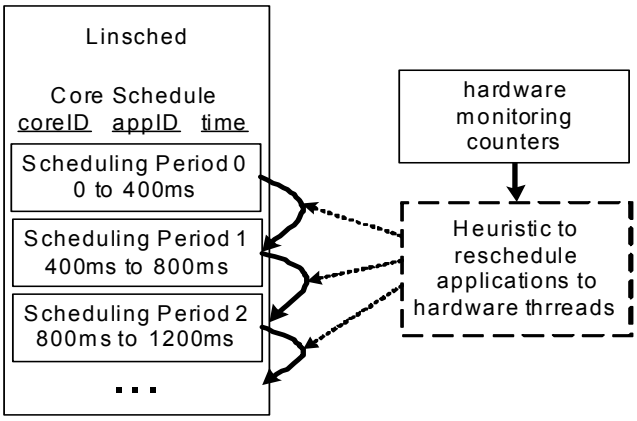

Figure 6. Communication between Linux Scheduler (Linsched) and hardware monitoring counters

4. SHARED CACHE-AWARE OS SCHEDULING – A CASE STUDY USING CMPSCHED$IM

When heterogeneous applications run simultaneously on a multi-core platform it is possible to have applications exhibiting the following different types of behaviors: 1) Poison applications – don't use the cache efficiently or have working sets much larger than the cache capacity available, they dominate the cache resource by excessively kicking out other applications data and may not benefit by using more cache 2) Neutral applications – largely insensitive to cache size and exhibit no change in performance irrespective of other co-running applications. Typically, these applications have very small working sets and 3) Vulnerable applications – applications that benefit from existing cache capacity and suffer in performance due to aggressive behavior of co-running applications. Today's platforms and OS have no visibility into application cache contention and its implications on either individual workload or overall system performance. Often, this lack of fine grained resource utilization information leads to poor individual performance as well as poor overall performance. Our primary goal was to monitor the shared cache usage of the applications and try and classify them as Poison, Neutral or Vulnerable applications so they can be scheduled by the OS/execution environment to minimize contention in the shared cache.

To illustrate the loss in performance due to sub-optimal scheduling, consider the scenario of 8 applications running on a 4 core system, with 2 Last Level Caches (LLCs) and 2 cores sharing the same cache. Fig 7a below shows a possible worst case scenario of poor scheduling of applications due to heavy contention in the shared cache. Through cache scaling characterization experiments, we have foreknowledge that swim and lucas are poison applications, while mcf and art are vulnerable applications. Ones that are neutral are eon and perlbmk while gcc and galgel also fall into the vulnerable category, but are vulnerable to a lesser degree than mcf and art. In the worst case scenario, mcf and art (vulnerable apps) end up running with swim and lucas (poison apps) and hence mcf and art would suffer heavy contention in the shared cache. A possible best case scenario is shown in Fig 7b, where mcf and art (vulnerable apps) are paired with eon and perlbmk (neutral apps). Though gcc and galgel are paired with swim and lucas, gcc and galgel are not affected as bad as mcf and art.

We performed some measurements in an Intel Core2 Quad core [4] system to study the performance impact on the worst case scenario compared to the best case and the results are shown in Fig 8.

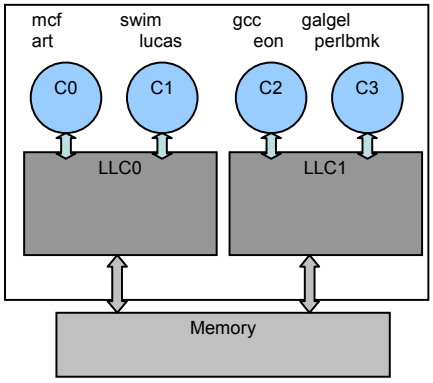

Figure 7a: Possible worst case scheduling scenario

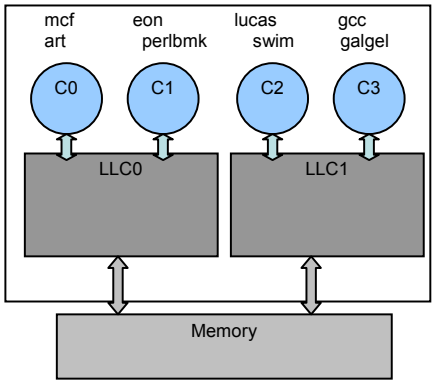

Figure 7b. Possible best case scheduling scenario

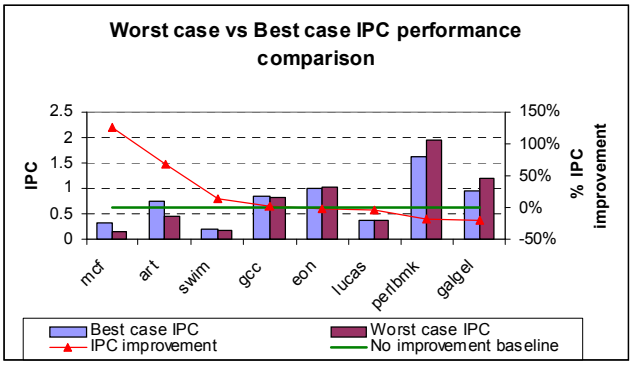

Figure 8. Worst case vs. Best case scheduling comparison

The primary Y-axis in Fig 8 shows the Instructions Per Cycle (IPC) throughput of each application and the secondary Y-axis shows the percentage IPC improvement per application. We can see that mcf and art showed a significant improvement in performance of as much as 125% and 70% respectively. Swim showed some slight

performance improvement and this is more due to the memory bandwidth than the cache as swim is not affected much by cache, and same is the case with lucas. Perlbmk and eon were not affected much and so is gcc. Galgel suffered some loss in the best case; however overall, the best case shows an average IPC improvement of 20%. Our first challenge is to come up with a heuristic that would achieve near optimal scheduling using existing counters like Misses Per Instruction (MPI) and potential future counters like capacity usage of each application in the shared cache. The bigger challenge also is to quantify and show through simulations, if such capacity counters were available, how efficient would our heuristics be if we were to use MPI alone, Capacity alone or both MPI and Capacity information. It should be noted that it is critical to model the right scheduling behavior in terms of how much time each application gets on each core because the length of times each application runs can also affect the performance. For example if the time an application gets on a core is 5ms vs. 50ms, then there would be an obvious effect on the shared cache performance due to cold start misses and evictions that entail when a new application comes in. This is one of the main reasons why it is important to incorporate a real scheduler like the Linux scheduler simulator Linsched that we employ in CMPSched$im.

In the simplest case, we started off with heuristics using capacity alone and MPI alone. Intuitively, it appears that a better way of scheduling applications is to make sure that high capacity applications don't run at the same time sharing the same LLC and it would make sense to pair high capacity applications with low capacity applications. Less intuitive but still plausible would be to make sure that high MPI applications don't co-run on the same LLC with other high MPI applications. In the absence of capacity counters, MPI can give a reasonable indication towards high capacity applications.

In order to understand the difference (and similarity) between the MPI and Capacity counters, we conducted a simple cache scaling study with the workloads. Fig 9 shows the cache scaling results from the experiments we performed on a present day CMP platform.

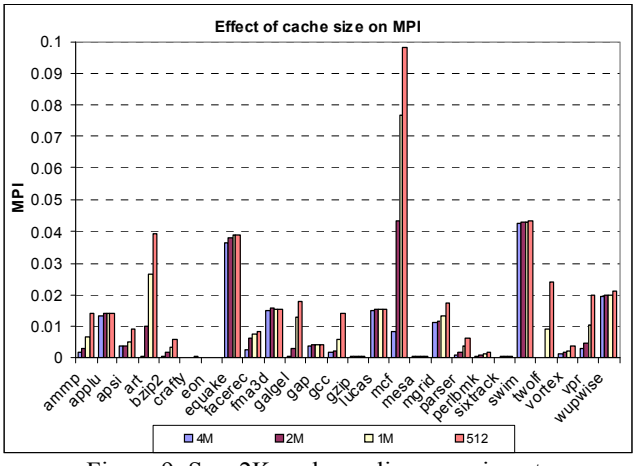

Figure 9: Spec2K cache scaling experiments

The cache was scaled down from 4M down to 512K. We can see that applications that have a high MPI (a miss every 100 instructions) are applications like swim, lucas, equake etc. From simulation studies we have done earlier, beyond the scope of this paper, we have foreknowledge that these are high capacity applications. However, MPI being an indicator of capacity is not always accurate because applications like mcf and art show a huge variation in MPI as we scale the cache size down. Next we will describe how MPI and capacity were used in our heuristics. The details about interactions among various modules in CMPSched$im to communicate the monitored information is described in section 3. In the following sub-sections we will be using the same workload mix and cache hierarchy as presented earlier in Fig 7a, b. CMPSched$im running at blazing speeds helped us complete full workload runs overnight when run on multi-core systems like the Core 2 Quad.

4.1. MPI Based Heuristic in CMPSched$im

In the MPI based heuristic, the MPI of all application are sampled every 800ms. The 800ms is a value that was chosen empirically to make sure each application has had enough time on the core. Also, the time on core for each application should be reasonably long to capture the behavior of the application and avoid fine grained phase changes. Once the MPI values are sampled, the applications are sorted ordering them in decreasing order of MPI and mapped to the cores as shown in Fig 10. The mapping is done by migrating the applications to appropriate cores using APIs provided with Linsched.

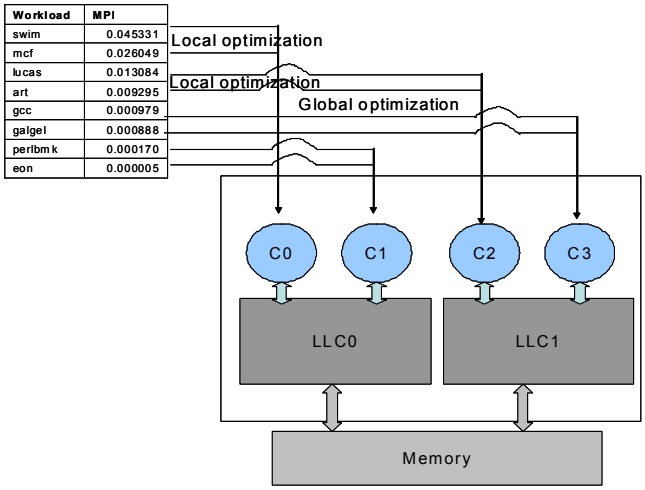

Figure 10. Mapping of applications to cores based on MPI Heuristic

The mapping is aimed at making sure that two high MPI applications don't co-run such that they share the same cache. It achieves this by local and global optimization as shown in Figure 10. With local optimization the two highest MPI applications are run on the same core so they are separated by the core because no two applications can run on the same core at the same time. With global optimization, the high MPI applications are mapped to cores such that they

run on different LLCs. When the applications are launched, they are started with affinity such that they start at the worst case as shown in Fig 7a. Ideally our heuristic should take the execution to the mapping in Fig 7b. Though the MPI heuristic achieves a reasonably better mapping, swim may end up running with perlbmk or eon and this is not ideal because, swim does not need a lot of cache (poison application) and pairing it with a low cache usage application like perlbmk or eon would be sub-optimal. There could be instances where the ideal mapping may be achieved, even with just MPI heuristic but based on what we observed this is not the case as the MPI heuristic does not do a good job in distinguishing poison applications (lucas and swim) from vulnerable applications (mcf and art).

4.2. CAPACITY BASED HEURISTIC IN CMPSCHED$IM

In the capacity based heuristic the capacity of each application in the shared LLC is sampled at the time, the application is switched out of the core. As noted earlier, an application's time on the core is determined by the schedule returned by Linsched. Unlike the MPI heuristic where the MPI is sampled at 800ms intervals, we have to sample capacity when an application is switched out because, the new application that comes in may evict the lines of the application that just got switched out. In the MPI heuristic at any time we know on a per application basis how many misses were incurred and how many instructions of that application were executed. Now we take an average of the capacity values for each application as an application may have run several times and switched out several times in 800ms. As before, the applications are sorted in order of decreasing average capacity, and mapped similar to what was done in Figure 10. The goal with the Capacity based heuristic is to again make sure that no two high capacity applications run at the same time sharing the same cache. This is achieved through local and global optimizations as pointed out in the previous section but this time using average capacity of each application during the 800ms sample run. Intuitively, the capacity based heuristic would achieve better results than the MPI because unlike huge MPI variations of vulnerable applications, there will not be huge variations in capacity. This would become clear as we examine the end performance results and migration patterns that will be described in the following sections.

4.3. CAPACITY AND MPI BASED HEURISTIC IN CMPSCHED$IM

While studying the performance of each application using MPI alone Heuristic and Capacity alone Heuristic, we noticed that both the Heuristics were not able to achieve near optimal scheduling and this reflected in their sub-optimal performance.

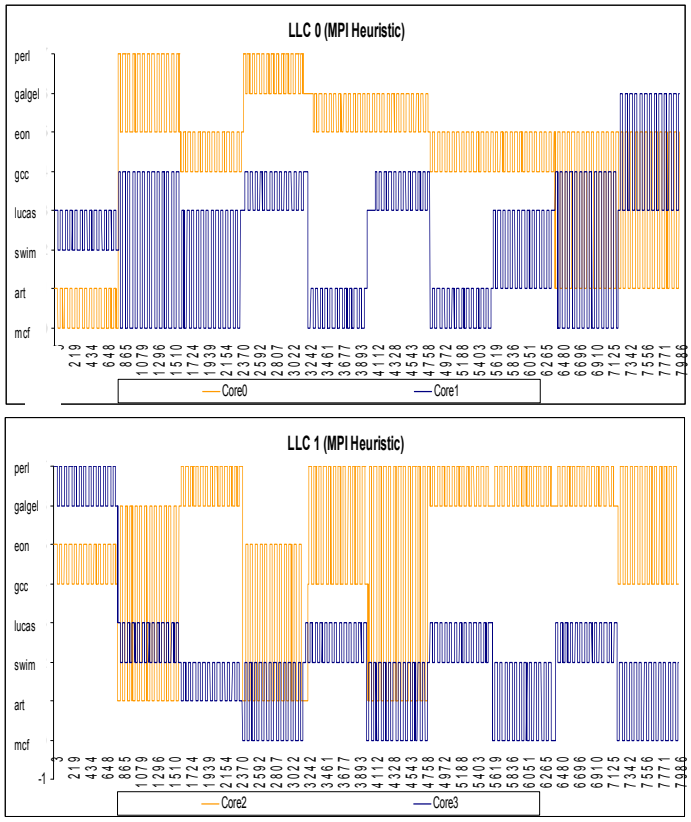

Figure 11. Application to core mapping for MPI Heuristic as reported by CMPSched$im

Figure 11 shows the application to core mapping for each of the LLCs. The X-axis is the elapsed time in milliseconds. The Y-axis shows the workloads and the series represent the core in which the application ran. We can see that in LLC0 up to 800ms art and mcf ran in core 0 while lucas and swim ran in core 1 sharing the same cache and this represents the worst case which was at launch. As the simulation progresses, the MPI heuristic had already kicked in at 800ms and the heuristic figures out a mapping based on the MPI. From 800ms onwards mcf and gcc are paired with eon and perlbmk though the ideal would have been mcf and art with eon and perlbmk. However mcf was separated from lucas and swim which are running in a different cache now with galgel and art. mcf's MPI drops low being paired with eon or perlbmk and hence the heuristic begins to deteriorate in efficacy. Also mcf and art, which are the most vulnerable applications migrate from LLC0 and LLC1 causing a ping-pong effect resulting in unwanted migration costs. Ideally we want the heuristic to arrive at the best case mapping and stay there. Figure 12 shows the migration patterns with capacity heuristic.

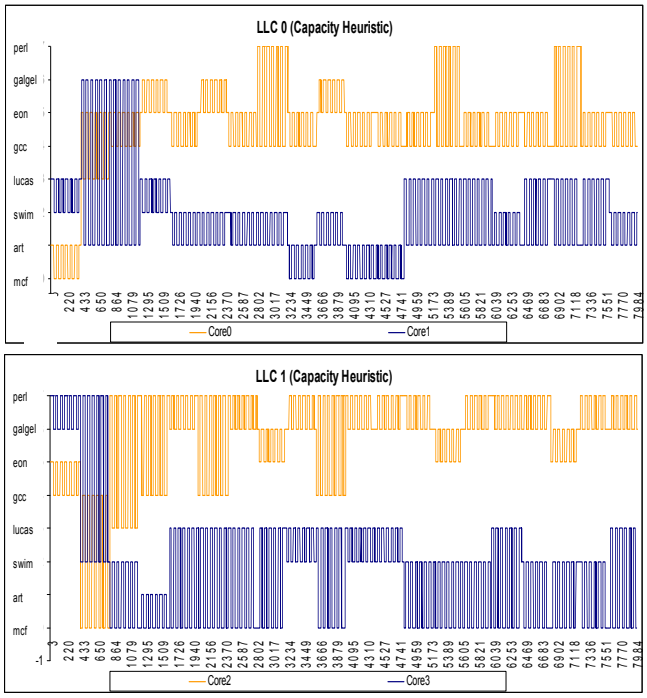

Figure 12. Application to core mapping for Capacity Heuristic as reported by CMPSched$im

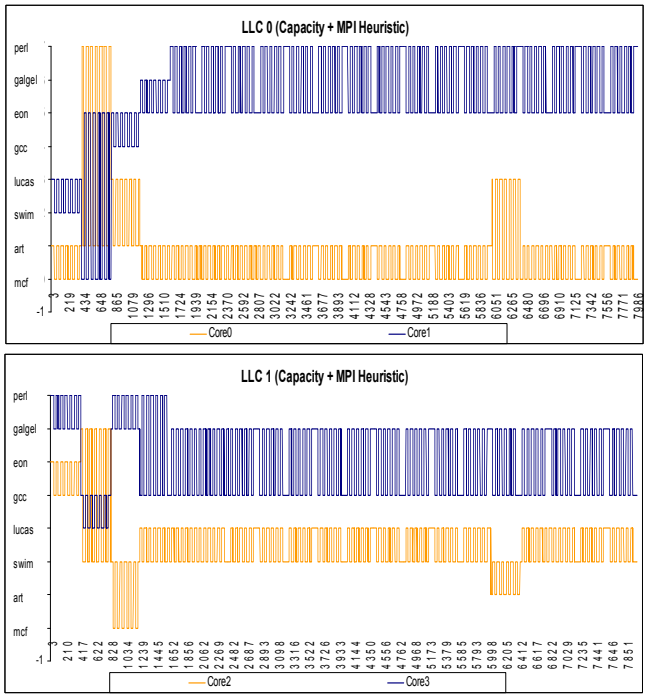

Figure 13. Application to core mapping for Capacity+MPI Heuristic as reported by CMPSched$im

It can be observed that the capacity heuristic is a lot more stable than the MPI heuristic in the sense that it did not bounce around the most vulnerable applications mcf and art from LLC0 to LLC1 back and forth as much as the MPI heuristic did, thereby minimizing the migration costs. The capacity heuristic also achieves a good local and global optimization. However, what the capacity alone and MPI alone heuristic lack in, is that they do not identify the poison applications. The hybrid heuristic that uses both capacity and MPI information to identify poison applications achieves near optimal scheduling. The following is how the heuristic is designed. Once the applications are sorted in order of decreasing capacity, the applications in the top half (highest capacity) are monitored to see which ones have high MPI. We chose various thresholds for the MPI for example an MPI of 0.01 (a miss every 100 instructions) or an MPI of 0.008 (more aggressive), and any high capacity application that has an MPI above this threshold is classified as poison applications. At the end of the first sampling point, the hybrid Capacity+MPI heuristic would not successfully distinguish poison applications (swim and lucas) from vulnerable applications (mcf and art), because mcf would have a high MPI as it started with worst case mapping. However as the simulation progresses through sampling points, the heuristic would almost always identify the poison applications as the MPI values of mcf and art would drop. Once lucas and swim are identified as poison applications, they are separated from mcf and art by global optimization (put in a different cache) and the lowest capacity applications perlbmk and eon are paired with mcf and art achieving the best case oracle mapping.

Figure 13 shows how the hybrid Capacity+MPI Heuristic achieves near optimal scheduling as what we want in the best case. mcf and art are almost always paired with eon and perlbmk. Also, the unwanted migration costs are also avoided. There are some minor hiccups as we see at second 6, but this is more of a workload phase issue than a heuristic issue. Now we will present the individual application performance and improvements with the heuristics in overall average performance improvement.

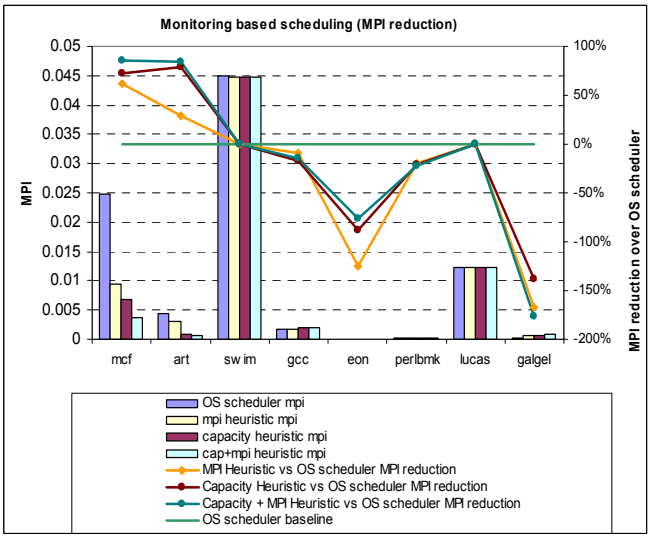

Figure 14a. MPI Performance comparison

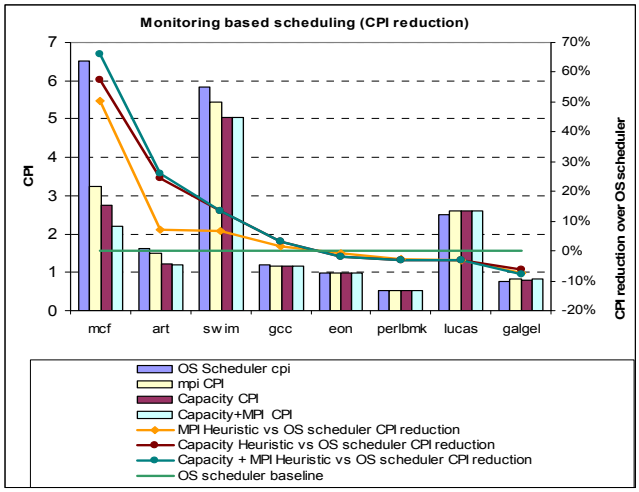

Figure 14b. CPI Performance comparison

Figure 14a shows the MPI for each application on the primary Y-axis and shows how each heuristic does in reducing the MPI thereby improving performance. The secondary Y-axis shows the percentage MPI reduction compared to the default OS scheduler. The default OS scheduler represents basically what the launch order is and hence in this case it corresponds to the worst case mapping. The Linux scheduler affinitizes each application to the core it was launched in, unless the applications are not balanced across cores. In this case, since there are 2 applications in each core, the load balancer does not kick in. We can see that mcf and art show significant drop in MPI with each of the above heuristics with MPI+Capacity heuristic doing best, followed by Capacity alone and MPI alone. Though eon and galgel show significant increase in MPI, the actual MPI values are so low, that the actual performance of the application is not affected that much as can be seen in Figure 14b that shows the CPI (Clocks Per Instruction) performance. With the Capacity+MPI scheduler mcf and art showed a reduction of over 60% and over 20% respectively. Galgel showed only less than 10% increase in CPI. Figure 15 shows the average of the percentage increase in IPC (Instructions Per Clock) for all the applications.

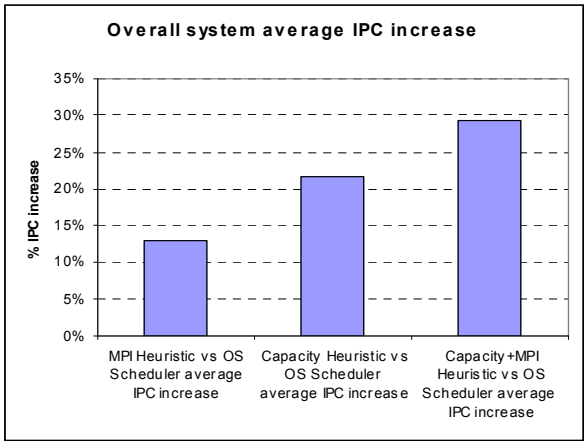

Figure 15. Average of percentage IPC increase of all applications

Comparing to the default OS scheduler, the MPI Heuristic achieved 12% increase while the Capacity Heuristic achieved 21% increase while the Capacity+MPI Heuristic achieved a huge improvement of 30%. We have experimented with other combinations of workloads and the Capacity+MPI Heuristic almost always arrives at the near optimal scheduling. The case study clearly shows how valuable CMPSched$im proved in not just helping us quantify the improvements in performance, but also helping us arrive at the hybrid Capacity+MPI Heuristic.

5. CONCLUSION AND FUTURE WORK

In this paper we discuss the complexity of shared resource management in CMP platforms and the need for optimizing the interactions between the OS/execution environment and the CMP platform hardware architecture. We present the limitations of existing simulation tools and motivate the need for alternate fast and flexible tools to explore better OS/CMP interaction. We propose the CMPSched$im simulation infra-structure to successfully overcome the limitations of execution and trace-driven simulations and accurately capture OS/execution environment and hardware interaction and this, with excellent speeds in the order of several MIPS. This enabled us to get results very quickly. We also illustrate with a case study how CMPSched$im helped us design a simple heuristic that uses present day cache miss, instruction retired counters and proposed future capacity counters to achieve near optimal scheduling. We compare the per application performance improvement and overall system performance improvement compared to the base case OS scheduler, MPI Heuristic, Capacity Heuristic and Capacity+MPI Heuristic. We also show how efficient the Capacity+MPI Heuristic is in achieving a significant overall performance improvement of up to 30%.

Future work in this area would first involve more enhancements to add more detailed micro-architectural models. We would also like to study a richer set of heterogeneous workloads and possibly enable other types of execution environment schedulers (virtual machine monitors for example). We expect that this approach will also help significantly speed up additional OS/CMP interaction exploration (e.g. heterogeneous cores, hybrid core+accelerator architectures, etc).

ACKNOWLEDGEMENTS

We would like to thank John Calandrino for his contributions with Linsched and supporting migration APIs that helped us with easy integration into the simulation infra-structure and also Jessica Young, Tong Li and, Dan Baumberger for their contributions to Linsched.

REFERENCES

[1] AMD Inc., "AMD Multi-core Processors," http://multicore.amd.com/en/

[2] D. Chandra, F. Guo, et al. Predicting inter-thread cache contention on a chip multiprocessor architecture", 11th International Symposium on High Performance Computer Architecture (HPCA), Feb 2005.
[3] N. Enright Jerger, D. Vantrease, M. H. Lipasti, "Evaluation of Server Consolidation Workloads for Multi-core Designs," IISWC-2007
[4] Intel Xeon 5400 Series, ftp://download.intel.com/products/processor/xeon/dc54kprodbrief.pdf
[5] Intel Corporation, "World's first quad-core processors for desktop and mainstream processors," http://www.intel.com/quad-core/
[6] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In Proc. of 18th Annual International Conference on Supercomputing (ICS'04), July 2004.
[7] R. Iyer, L. Zhao, et al., "QoS Policies and Architecture for Cache/Memory in CMP Platforms", the ACM SIGMETRICS
Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), June 2007
[8] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In Proc. of 13th Int'l Conf. on Parallel Arch. & Complication Techniques (PACT), Sept 2004.
[9] R. Kumar, D. Tullsen et al., Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance, In 31st International Symposium on Computer Architecture, June, 2004
[10] M. R. Marty, M.D. Hill Virtual Hierarchies to Support Server Consolidation, ISCA 2007
[11] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches In Proc. of Annual Int'l Symposium on Microarchitecture (MICRO), June 2006.
[12] N. Rafique, W.T. Lim and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In Proc. of the 15th International Conference on Parallel Architectures and Compilation Technology (PACT 2006), Sept 2006
[13] L. Zhao et. al Cache Scouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms, PACT 2007
[14] Pin – A dynamic binary instrumentation tool http://rogue.colorado.edu/pin/
[15] A. Jaleel et al: CMP$im- A Pin-Based On-The-Fly Multi-Core Cache Simulator, Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS) co-located with ISCA 2008
[16] J. Calandrino et al: Linsched-The Linux scheduler simulator, PDCCS 2008