A Case for Integrated Processor-Cache Partitioning in Chip Multiprocessors*

Shekhar Srikantaiah, Reetuparna Das, Asit K. Mishra, Chita R. Das and Mahmut Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{srikanta, rdas, amishra, das, kandemir}@cse.psu.edu

ABSTRACT

Existing cache partitioning schemes are designed in a manner oblivious to the implicit processor partitioning enforced by the operating system. This paper examines an operating system directed integrated processor-cache partitioning scheme that partitions both the available processors and the shared cache in a chip multiprocessor among different multi-threaded applications. Extensive simulations using a set of multiprogrammed workloads show that our integrated processor-cache partitioning scheme facilitates achieving better performance isolation as compared to state of the art hardware/software based solutions. Specifically, our integrated processor-cache partitioning approach performs, on an average, 20.83% and 14.14% better than equal partitioning and the implicit partitioning enforced by the underlying operating system, respectively, on the fair speedup metric on an 8 core system. We also compare our approach to processor partitioning alone and a state-of-the-art cache partitioning scheme and our scheme fares 8.21% and 9.19% better than these schemes on a 16 core system.

1. INTRODUCTION

Chip multiprocessors (CMPs) have become the de facto microprocessor architectures and have received strong impetus from all leading chip manufactures. The advent of CMP era has resulted in a sustained emphasis on the need for development of techniques to extract thread-level parallelism. Given the limited ability of current compilers to extract a large fraction of the available parallelism, we envision a typical workload on future multicore processors to consist of a dynamic and diverse range of applications that exhibit varying degrees of thread level parallelism. Emergence of server consolidation and myriads of multimedia applications further emphasize a resource sharing based usage model, which will persist in future systems.

Interference in a multi-programmed workload exists on a coarse scale like contention for shared last level caches [5, 6, 12, 31], memory bandwidth and most importantly the available computing resources or processors. As the number of cores and associated shared platform resources on die keep increasing, there is an urgent need for efficient, integrated shared resource management policies for CMPs. A flurry of recent research has addressed partitioning and/or scheduling for various shared resources like processors [36, 4, 9], shared last level cache [27, 32, 6, 26, 17] and memory bandwidth [28, 22] individually. However, the focus on integrated shared resource management schemes has been rather modest [2, 8, 24, 33]. To the best of our knowledge, there has been no prior work on integrated partitioning of processors and shared last level cache among multiple concurrently executing multithreaded programs. This work proposes that the partitioning of the shared on-chip cache (L2) be done in a manner that is aware of the processor partitioning.

Although shared cache partitioning has been reasonably well studied recently, processor partitioning in the context of providing insulation to multiple multithreaded applications on CMPs has not been well studied. When we say processors of a CMP are "partitioned", we mean that processors are assigned to individual applications in such a way that all threads of a particular multithreaded application are affinitized to execute on a restricted set of processors. An example scenario of both processor and cache partitioning on an eight processor CMP executing four multithreaded applications is depicted in Figure 1. In current practice, processor partitioning and cache partitioning are decoupled as the processor partitioning is explicitly or implicitly handled by the operating system, while cache partitioning is generally handled by the micro-architecture.

In a traditional scheduling approach, the processor partitioning is implicit and the assigned "partitions" change after every time quanta. It is important to note that fairness among threads is generally an important metric in such an implicit partitioning. In this paper, we propose to use a more explicit processor partitioning strategy and leverage the knowledge of such an explicit processor partitioning to select the most appropriate on-chip cache partitioning. Note that, we do not re-thread or change the number of threads of any application, but redistribute the threads on partitions of different sizes and facilitate the processor partition to accommodate all threads of the application.

*This research is supported in part by NSF grants CNS #0720645, CCF #0811687, CCF #0702519, CNS #0202007 and CNS #0509251, a grant from Microsoft Corporation and support from the Gigascale Systems Research Focus Center, one of the five research centers funded under SRC's Focus Center Research Program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC09 November 14-20, 2009, Portland, Oregon, USA
Copyright 2009 ACM 978-1-60558-744-8/09/11 ...$10.00.

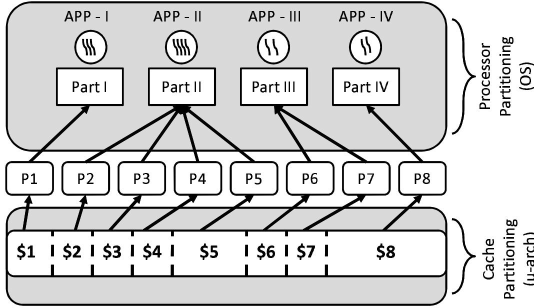


Figure 1: A typical scenario of processor and cache partitioning in a CMP with 8 processor cores and a shared last level cache executing four applications.

Architectural support for software controlled cache partitioning [27] has been studied in the past. However, including [27], none of the cache partitioning schemes that we are aware of leverages the knowledge of the processor partitioning in order to provide stronger isolation. Specifically, the cache partitioning is per processor (*i.e.,* each cache partition is associated with a processor) in most of the previously proposed cache partitioning schemes [32, 6, 26, 17]. We propose to partition the cache by allocating one partition to each processor-set (processor partition), thereby encouraging constructive sharing among threads of the same application and alleviating the impact of conflict misses in the cache due to the impact of interactions between memory references of different applications in the cache. Therefore, the cache partitioning happens in a manner that is aware of existing processor partitions. Further, the proposed partitioning approach is *iterative*, wherein partitioning of processors and cache happen in a series of iterations. The processor partitioning in one iteration influences the cache partitions in the same iteration and the cache partition at the end of the current iteration influences the processor partitioning to be selected dynamically in the following iteration. We refer to such an iterative processor partitioning aware cache partitioning scheme as *integrated processor-cache partitioning*. We believe that such an adaptive approach is necessary to provide meaningful service differentiation to a consolidated set of applications running on a single platform. The goal of our integrated partitioning approach is to dynamically partition the resources so as to benefit majority of applications (fairness) without hurting performance of any one application (QoS) by more than a threshold percentage. The merits of the paper are the following:

- We argue that partitioning of either the shared L2 cache or the processors in isolation does not generally lead to optimal system performance and show that an operating system enforced, integrated processor-cache partitioning policy can provide effective performance isolation and improve overall system speedup across multiple concurrently executing multithreaded applications.
- We study the achievable benefits from an integrated partitioning scheme using a regression based model to predict the behavior of applications and suggest techniques to prune the large search space in order to find the most suitable processor and L2 cache partitions. Our approach improves the system throughput and fairness across multiprogrammed workloads, while guaranteeing quality of service (QoS) with equal share of processors and cache as the baseline.
- We also show that fair partitioning is not necessarily the best allocation scheme and that allocation policies need to dynamically react to the varying degrees of parallelism and growing or shrinking demands (on both CPU and memory) exhibited by the workloads during different program execution phases.

2. BACKGROUND AND MOTIVATION

In the presence of contention and varying demands on the system, providing system load-independent performance guarantees to applications becomes crucial apart from improving the overall throughput/performance of the workload. Several metrics have been suggested in the literature that quantify the effectiveness of resource partitioning schemes in terms of improving the overall throughput, utilization [25], fairness [17, 19], and quality of service (QoS) [11, 13, 14] of the multiprogrammed workloads.

2.1 Metrics of Interest

In this paper, we use three metrics (similar to those used in [6] and [26]): (1) A weighted speedup metric (WS) that quantifies the improvement in overall system throughput (across different applications of the multiprogrammed workload); (2) A fair speedup metric (FS) that balances both performance and fairness; and (3) A QoS metric that measures the per application performance degradation with respect to the baseline (equal resource share) case. These metrics are defined as follows:

The Weighted Speedup metric (WS) of the workload using a partitioning scheme is defined as the sum of per application speedups (gain in instructions per cycle) using the scheme with respect to the baseline equal resource share case.

$$WS(scheme) = \sum_{i=1}^{N} \frac{IPC_{app_i(scheme)}}{IPC_{app_i(base)}}, \quad (1)$$

where N is the number of applications in the workload. The base case for our weighted speedup metric is the performance of individual applications with equal partition as opposed to single partition in [26]. Since WS(*base*) $= \sum_1^N 1 = N$, we define normalized weighted speedup as ratio of WS to N. The weighted speedup metric can be misleading as an objective metric because it can be optimized by disproportionately favoring a single application, thereby hurting fairness and QoS. Therefore, we use normalized weighted speedup for comparison purposes only and not as a metric to search for the most suitable partition.

The Fair Speedup metric (FS) of the workload using a partitioning scheme is defined as the harmonic mean of per application speedup using the scheme with respect to the baseline equal resource share case.

$$FS(scheme) = N / \sum_{i=1}^{N} \frac{IPC_{app_i(base)}}{IPC_{app_i(scheme)}}, \quad (2)$$

where N is the number of applications in the workload. Note that, the harmonic mean is computed over per-application speedup as opposed to per-thread speedup in [6], as our workload is constituted of multiple multithreaded applications and not single threaded applications. Therefore,

$$IPC_{app_i(scheme)} = \sum_{k=1}^{P} IPC_{proc_k}, \quad (3)$$

where P is the number of processors in the partition allocated to app_i. Note that, FS is an indicator of the overall improvement in IPC gained across the applications. It is also a metric of fairness as the harmonic mean of a vector is maximized when all the elements of the vector are equal. In this work, FS is our main objective metric, *i.e.*, our objective is to find a processor/cache partition that yields the highest FS for the workload, subject to performance guarantee conditions described below.

The QoS metric of the workload using a partitioning scheme is defined as the summation of per-application slowdowns (if any) using the scheme over the baseline equal resource share case.

$$QoS(scheme) = \sum_{i=1}^{N} min(0, \frac{IPC_{app_i(scheme)}}{IPC_{app_i(base)}} - 1), \quad (4)$$

where N is the number of applications in the workload. By keeping this metric bound within a threshold value QoS_{Th} (negative percentage), the guaranteed performance for the application is also bound to be within the threshold value from the baseline equal resource share case. In formal terms, our objective is to find the most suitable processor/cache partition P_{opt}, such that, $FS(P_{opt}) \geq FS(P_{nopt})$, for any partition P_{nopt} and $QoS(P_{opt}) \geq QoS_{Th}$. Note that the criterion on QoS is a bound on the minimum value of $QoS(P_{opt})$ as $QoS(P_{opt}) \leq 0$. A more elaborate and formal discussion on these metrics is presented in [7, 30, 15]. Similar metrics have also been used in [17, 23, 35, 6, 26].

2.2 Static Partitioning

The default timesharing scheduling in most operating systems attempts to allow each process on the system to have relatively equal or fair CPU access (unless the processes are explicitly prioritized). This might not be the most appropriate way of "partitioning" the resources when the variation in the degree of thread level parallelism is high among the processes. A more explicit partitioning that provides each application with a processor partition proportional to its thread level parallelism augmented with a good cache partition could potentially accentuate the benefits obtained by increasing the overall system throughput. Fairness and QoS can further be guaranteed by selecting the partitions that meet the QoS threshold for maximizing the fair speedup across all the processes. Note that, our processor partitioning scheme changes the affinity of all threads of an application to a restricted set of processors, but does not entirely replace the default timesharing scheduler of the underlying operating system. After our partitioning scheme allocates a certain share of processors to an application, the individual threads of the application are scheduled independently by the underlying OS scheduler, albeit restricted to the partition assigned to the process. Such a partitioning scheme based on modifying the affinity of applications to processor sets not only simplifies the implementation but also reduces the overheads associated with re-partitioning as the migration necessary for re-partitioning happens at a coarser granularity. Note that the partitioning can be *static* where the partitioning remains unaltered for the period of execution of a workload or be *dynamic* where partitions are modified dynamically during execution. The overheads associated with re-partitioning becomes crucial in a dynamic partitioning scheme like ours.

The combinatorial explosion of the number of configurations to be considered for an exhaustive search of all possible static partitions (mapping between processes and processor sets) is evident. For example, the number of possible processor partitions of 8 processors for 4 applications is 35 and that for 16 processors and 8 applications is 116280. In fact, the number of compositions (ordered partitions) of p processors for k applications is given by $C(p-1, k-1)$, which is of manageable complexity for 8-16 processors but is excessively high for values of p beyond 32. The numbers only grow larger when cache partitioning is also considered. Therefore, it is impractical to evaluate all partitions before deciding on the most suitable one. A performance predictor that predicts the performance of applications under a given share of processors (or cache while keeping the other constant) would help to reduce the number of actual configurations to be evaluated before making the decision. We also attempt to shrink this search space by making some key observations about the two metrics, namely, the fair speedup metric and the QoS metric and the correlation between them. We proceed by discussing these performance trends in more detail.

2.3 Performance Trends

Multiprogrammed workloads exhibit different kinds of variability in their behavior. Variability in terms of the degree of maximum parallelism is observed on the larger scale. However, programs also exhibit a lot of variation in the computational demand and memory access characteristics at the granularity of program phases that typically last for a fraction of the total execution time of the program. This leads to significant variations in performance metrics with varying processor and cache allocations.

Figure 2 (a) plots a surface graph of the variations in the fair speedup metric for a multiprogrammed workload consisting of apsi, art, applu, and ammp (all from the SPEC OMP 2001 benchmark suite) with various static processor and cache partitions on a 8 processor system with 16MB shared L2 cache, with respect to the baseline case of equal resource partitioning (2 processors and 4MB L2 cache per application). We can make important observations from the data presented in Figure 2 (a).

- The performance of applications, as indicated by the fairness metric, is highly sensitive to the processor and cache partitions assigned to them. Due to this high sensitivity, it is critical to select the right partition, and the price of a wrong selection can be very high, as indicated by a few spikes in the fair speedup metric (in black) and the difference between the fair speedup metric of the peaks and the lower parts of the plot (below a fair speedup metric value of 1, shown in grey in the graph).
- The impact of partitioning processors is more pronounced on the fair speedup of applications than L2 cache partitioning as indicated by relatively lower levels of variation along the axis of cache partitions. We use this knowledge and adopt an iterative policy that first partitions the processors followed by cache partitioning and iteratively proceeds to tune the configuration (one at a time), thereby reducing the complexity of searching the configuration space.

2.3.1 *Correlation Between FS and QoS Metrics*

Figure 2 (b) plots a surface graph of the variations in QoS metric for different static partitions. All the observations made with respect to the fair speedup metric also hold for the QoS metric. It is clear from the graph that,

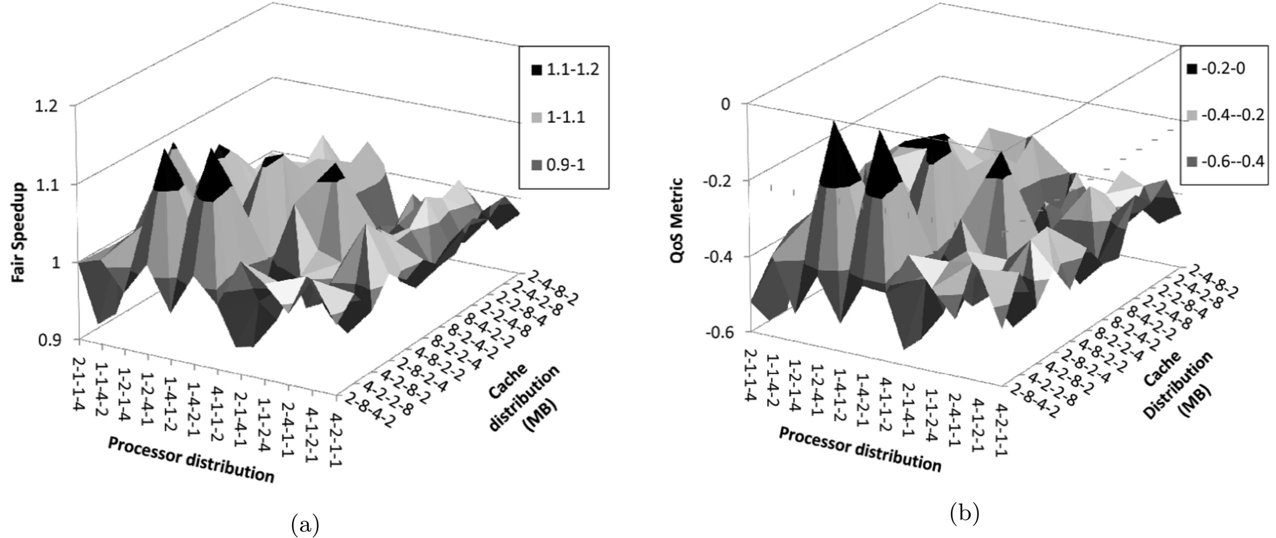


Figure 2: **(a) Fair speedup metric and (b) QoS metric for different static partitions of processors and shared L2 cache for a multiprogrammed workload consisting of *apsi, art, applu* and *ammp* on a 8 processor CMP with a 16 way associative, 16 MB shared L2 cache. Each partition (p1, p2, p3, p4) on the processor partition axis and (c1, c2, c3, c4) on the cache partition axis represents a partition of the 8 processors and 16MB cache to apsi, art, applu, and ammp respectively.**

in case of static partitioning of processors and L2 cache, there is very good correlation between the QoS metric and the fair speedup metric (the measured correlation coefficient between the two metrics in this case is 0.92 and on an average across different workload mixes is 0.84). Similar observations about the correlation between FS and QoS metrics have also been made in [10]. A consequence of this correlation is that a partition that leads to higher QoS (lesser magnitude of the negative value) also leads to a higher value of the fair speedup metric across applications. But, what is important to note here is that, while a small degradation in the fair speedup metric (e.g., below 1) is tolerable (although not preferred as we would like to maximize the fair speedup metric), we cannot tolerate any degradation above a certain threshold (QoS_{Th}). Therefore, we can use the QoS metric to filter the partitions that do not meet the QoS requirement and be rest assured that we are not eliminating any potential candidates for optimization with respect to the fair speedup metric.

3. INTEGRATED PARTITIONING

Deductions based on crucial observations from the previous section about an efficient integrated processor-cache partitioning scheme can be summarized as follows:

• An iterative partitioning policy involving partitioning of processors followed by cache partitioning in each iteration can be adopted to reduce the complexity of searching the configuration space. One of the important ramifications of this decision is the flexibility to use any software or hardware based cache partitioning scheme along with our processor partitioning scheme.

• We can use the QoS threshold criterion for eliminating a number of partitions that do not meet our requirements. For a dynamic partitioning scheme, such shrinking of search space and local optimal configuration selection should be done over small epochs. The epochs should be small enough to not forsake any opportunities to adapt to dynamic phase changes but large enough to keep the overheads to a minimum.

• A predictor that predicts the performance (IPC) of each application for all possible processor shares (or cache shares) keeping the cache share (or processor share) equal among the applications is necessary for searching the most suitable configuration and for reducing the number of configurations we need to evaluate as it is impractical to evaluate all possible partitions.

Note that, although we are predicting the performance (IPC) of each application for all possible processor shares (or cache shares), our goal is to compute the fair speedup metric and QoS metric of the multiprogrammed workload for various candidate partitions and select the most suitable partition that maximizes the fair speedup metric within the bounds of the QoS metric threshold. IPC is a metric that characterizes program execution and can be easily obtained online to perform online optimizations like dynamic processor and cache partitioning. Note that metrics like execution time cannot be easily obtained online at different intervals of time during program execution in order to perform optimizations like efficient resource management.

3.1 Search Space Pruning

We are using the QoS metric, specifically, to set a threshold on the maximum degradation tolerable (as a negative percentage) to the application and to eliminate partitions that do not guarantee this level of performance. We would like to find an efficient way of pruning the search space of possible partitions using the knowledge about how it is derived. Notice that, an important property of the QoS metric, as seen from Equation (4) is that, it is a linear summation of the function $min\left(0, \frac{IPC_{app_i(scheme)}}{IPC_{app_i(base)}} - 1\right)$ evaluated for each i in the range of 1 to N, where N is the number of ap-

Algorithm 1: INTEGRATED_PART(P, C, k, QoS_{Th})

/*Partitions P processors and C cache ways to k applications.
Maximizes FS and maintains QoS bounded within QoS_{Th}*/

//Main iteration performing integrated partitioning

while *true*
- PROC_PART(P, k, QoS_{Th})
- SLEEP($Proc_EnforcementInterval$)
- $P_C_Enf_Ratio \leftarrow \frac{Proc_EnforcementInterval}{Cache_EnforcementInterval}$
- **for** $i \leftarrow 1$ **to** $P_C_Enf_Ratio$
  do
  - CACHE_PART(C, k, QoS_{Th})
  - SLEEP($Cache_EnforcementInterval$)

plications. We denote this function using $\psi(i)$. Note that, $\psi(i), \forall i$ and QoS_{Th} are negative. The necessary condition for a partitioning scheme to be considered valid can then be represented as:

$$QoS_{Th} \leq \psi(1) + \psi(2) + \psi(3) + \ldots + \psi(N) \quad (5)$$

$$\Rightarrow QoS_{Th} \leq \psi(1) + \psi(2) + \psi(3) + \ldots + \psi(t), \quad \forall t, \text{ such that } 1 \leq t \leq N. \quad (6)$$

This implies that QoS_{Th} must be lesser than any partial sum of $\psi(i)$'s. Now, suppose we have an accurate predictor that predicts the IPC for each application i, when it is given a particular share of processors (or cache). We can easily evaluate $\psi(i)$ for all possible processor shares. If QoS_{Th} turns out to be greater than $\psi(i)$ evaluated for any particular processor share P_s, we can drop all partitions that assign P_s to application i. We call this process *partition space pruning*. We may continue to perform partition space pruning by evaluating partial sums of $\psi(i)$s to eliminate all combinations of P_ss, but this is typically not necessary unless the number of processors is greater than 32.

3.2 Iterative Integrated Partitioning

The integrated processor cache partitioning approach is *iterative* in nature. The partitioning of processors and cache happens in a series of iterations as shown in Algorithm 1. Processor partitioning in one iteration influences the cache partitions in the same iteration and the cache partition at the end of the current iteration influences the processor partitioning to be selected dynamically in the following iteration. This iterative execution of our approach enables (i) adaptive behavior of both processor and cache partitioning approaches to dynamic behavior of workloads; and (ii) inter-adaptability between processor and cache partitioning. The regression analysis based performance predictor used to perform processor and cache partitioning is described in the following sections.

3.2.1 Regression Analysis Based Prediction

One simple, yet interesting property that we exploit for predicting performance of applications on any share of processors is that the performance of almost all applications increases with the increase in their share of processors up to a threshold (limited by the maximum number of threads in the application), albeit with different rates of increase for different applications, depending upon their resource demands. Note that the thresholds themselves can be different for different applications.

Regression analysis [29] is a powerful and established tool to find the best fitting curve that fits a set of measured values. This method of finding the best fitting regression curve is called the least squares method because it minimizes the sum of the squared errors in prediction. In other words, let Y' be the predicted value of Y. Then, the least squares method minimizes $\sum(Y - Y')$. The quantity $(Y - Y')$ is the amount of error in prediction. It is the difference between the actual value Y and the predicted value Y'. Note that because the least squares regression curve minimizes the sum of the squared errors, it gives greater accuracy in prediction than any other possible regression curve. Another important property of regression is that the best fit curve can be computed incrementally from a set of measured values. The more values we have, the more accurate is the prediction. However, we can start predicting with as few values as necessary. Note also that, in our case, we are predicting the performance of applications given different amounts of processor/cache shares. Therefore, the prediction occurs in the full domain of possible partitions, *i.e.*, before any search space pruning.

Algorithm 2: PROC_PART(P, k, QoS_{Th})

comment: Partitions P processors to k applications.
Maximizes FS and maintains QoS bounded within QoS_{Th}.

for $i \leftarrow 1$ **to** k
//Enforce and get performance numbers from equal partition
do $EqualPart[i] \leftarrow P/k$
ENFORCE_PART($EqualPart$)
SLEEP($Proc_EnforcementInterval$)
GET_IPCS($EqualIPC$)

for $i \leftarrow 1$ **to** k
//Enforce and get performance numbers from alternate partition
do
- **if** $i mod 2 = 0$
- **then** $AlternatePart[i] \leftarrow (P/k) - 1$
- **else** $AlternatePart[i] \leftarrow (P/k) + 1$

ENFORCE_PART($AlternatePart$)
SLEEP($Proc_EnforcementInterval$)
GET_IPCS($AlternateIPC$)

$All_IPCs \leftarrow$ BOOT_REGRESSION($EqualIPC, AlternateIPC$)

//Main iteration performing processor partitioning

while *true*
- $Opt_Part \leftarrow$ FIND_OPTIMAL(All_IPCs, QoS_{Th})
- ENFORCE_PART($OptPart$)
- SLEEP($Proc_EnforcementInterval$)
- GET_IPCS($MeasuredIPC$)
- $All_IPCs \leftarrow$ REGRESSION($MeasuredIPC$)

3.2.2 Processor Partitioning

The number of possible processor shares that each application may receive grows with the number of processors. The goal of our regression based prediction is to predict performance values (IPCs) of each application i, denoted as $\{{\rho_i}^1, {\rho_i}^2, {\rho_i}^3 \ldots {\rho_i}^n\}$ for all possible processor shares $1 \ldots n$, given a subset of measured values $\{{\rho_i}^{s_1}, {\rho_i}^{s_2}, \ldots {\rho_i}^{s_m}\}$, where the shares $s_1, s_2, \ldots s_m \in \{1..n\}$.

In order to bootstrap such a regression based prediction, we need measured performance values (${\rho_i}^{s_{ei}}$ and ${\rho_i}^{s_{ai}}$) for at least two different processor shares $\{s_{ei}$ and $s_{ai}\}$ for each application i. Therefore, we need two partitions, $\{s_{e1}, s_{e2}, \ldots s_{ek}\}$ and $\{s_{a1}, s_{a2}, \ldots s_{ak}\}$, among the k applications such that $\sum_1^k s_{ei} = \sum_1^k s_{ai} = P$, where P is the total number of processors and $s_{ei} \neq s_{ai}, \forall 1 \leq i \leq k$. We provide these values to the predictor by evaluating only two partitions, independent of the number of applications in the system. The first partition, we call the *EqualPart*, is the partition of the available P processors into equal shares among the k

applications[1], *i.e.*, $\{s_{e1} = s_{e2} = \ldots = s_{ek} = P/k\}$. If k does not divide P, the additional (P mod k) processors are distributed among the first (P mod k) partitions. This is good enough, as our goal is to get two distinct mutually disjoint partitions. The second partition that we call *AlternatePart*, is the partition formed by alternately decreasing and increasing each s_{ei}, such that $s_{ei} \neq s_{ai}, \forall 1 \leq i \leq k$ (assuming an even number of processors, which is true in general). Both EqualPart and AlternatePart are used to bootstrap the regression based prediction of IPCs. Note that all partitions are enforced by the call to *Enforce_Part* followed by a sleep for a duration of $Proc_EnforcementInterval$. This interval parameter also decides the frequency at which partitions are reconfigured. IPCs of all the applications given the EqualPart and AlternatePart are determined by a call to the module *Get_IPCs*. Regression can further be used to determine IPCs of all applications given any processor shares. After this is determined for the first time, the partitioner enters a *monitoring loop*. In this loop, the routine FIND_OPTIMAL is invoked to find the partition that maximizes the fair speedup metric within the bounds of QoS_{Th} using search space pruning discussed earlier in Section 3.1. The newly found partition is enforced and the loop continues. Note that only the main loop of the processor partitioning logic executes each time $PROC_PART$ is called by the integrated partitioning scheme and bootstrapping needs to be done only once. An algorithmic description of our processor partitioning approach is shown in Algorithm 2.

We observed a sample window of 1 Billion cycles during the execution of a multiprogrammed workload consisting of apsi, art, barnes, and ocean using Algorithm 1 to partition an 8 processor CMP with 16MB L2 cache and an enforcement interval of 10 Million cycles (both processor and cache). The resultant plots of the dynamic partitions selected at every interval of 10 Million cycles is shown in Figure 3. We observe that our algorithm indeed adapts dynamically to the changing application behaviors and repartitions resources as necessary. It is important to observe from Figure 3 that, on an average, a processor re-partitioning only happens once in 65 Million cycles, which is much larger than the enforcement interval (10 Million cycles) used, although there are cases when the re-partitioning happens within 20 Million cycles. On the other hand, cache repartitioning occurs almost once in every 10M cycles. Therefore, it is important to select an appropriate enforcement interval that strikes the right trade-off between benefits of repartitioning and overheads of frequent repartitioning.

This brings us to the important topic of overheads involved in repartitioning processors and cache using our approach. There are perceivably two kinds of overheads involved with our approach: (i) The cost of regression analysis based prediction of the configuration to be enforced and (ii) The cost of repartitioning either the processors or the cache. We use a simple low overhead least-squares regression method for finding the optimal IPC. This corresponds to FIND_OPTIMAL () subroutine of Algorithm 2 and on average used around 2000 cycles for our experiments which is an insignificant fraction (0.02%) of the interval at which it executes. In general, the model has complexity that is proportional to the number of resources and number of applications. For our setup, the number of resources was set to 8 (CPUs) and 16 (cache ways) while the number of applications was 4. The overheads due to system calls to create processor partitions and bind applications to them is minimal due to coarser granularity of processor partitioning (comparable to scheduler time slice). Our OS interface for enforcement of cache partitioning is similar to that in [27]. A detailed discussion about timing and area overheads of the necessary hardware implementation can be found in [27]. *We would like to emphasize here that all the experimental results discussed in Section 5 includes each of these overheads as we perform our simulations on a full system simulator.*

Algorithm 3: CACHE_PART(C, k, QoS_{Th})

```
comment: Partitions C cache ways to k applications.
 Maximizes FS and maintains QoS bounded within QoS_Th.

for i ← 1 to k
 //Enforce and get performance numbers from equal partition
  do EqualPart[i] ← C/k
ENFORCE_PART(EqualPart)
SLEEP(Cache_EnforcementInterval)
GET_IPCS(EqualIPC)

for i ← 1 to k
 //Enforce and get performance numbers from alternate partition
     { if imod2 = 0
  do {  then AlternatePart[i] ← (C/k) − 1
     {  else AlternatePart[i] ← (C/k) + 1
ENFORCE_PART(AlternatePart)
SLEEP(Cache_EnforcementInterval)
GET_IPCS(AlternateIPC)

All_IPCs ← BOOT_REGRESSION(EqualIPC, AlternateIPC)

//Main iteration performing cache partitioning
            { Opt_Part ← FIND_OPTIMAL(All_IPCs, QoS_Th)
            { ENFORCE_PART(OptPart)
            { SLEEP(Cache_EnforcementInterval)
while true  { GET_IPCS(MeasuredIPC)
            {
            { All_IPCs ← REGRESSION(MeasuredIPC)
```

3.2.3 Cache Partitioning

Implementation of the algorithm for cache partitioning within each iteration of the integrated processor cache partitioning scheme can be done using the algorithm for processor partitioning (Algorithm 2) or we can choose to have any existing cache partitioning scheme modified to partition the cache among the chosen processor partitions instead of individual processors. For sake of uniformity and to avoid additional overheads of dual bookkeeping, we have chosen to use the regression analysis based algorithm explained above for cache partitioning as well. The modified algorithm for cache partitioning is depicted in Algorithm 3. Note that we use a saturating cache performance model represented by an *exponential decay function* of the cache size. Apart from this, the changes that are necessary in Algorithm 2 above for introducing L2 cache partitioning are (i) regression being bootstrapped with two different cache configurations and (ii) introduction of cache partitioning enforcement similar to processor partitioning into the monitoring loop (iteration) after the partitioning of processors. *It is important to note that in each iteration of the integrated processor cache par-*

[1] Note that, we are assuming here that $k < P$. This condition is also necessary in order to have a mutually disjoint partition of P processors among k applications. However, if this condition is not true, we can adopt clustering schemes and treat applications that display *symbiotic* interaction as one unit.

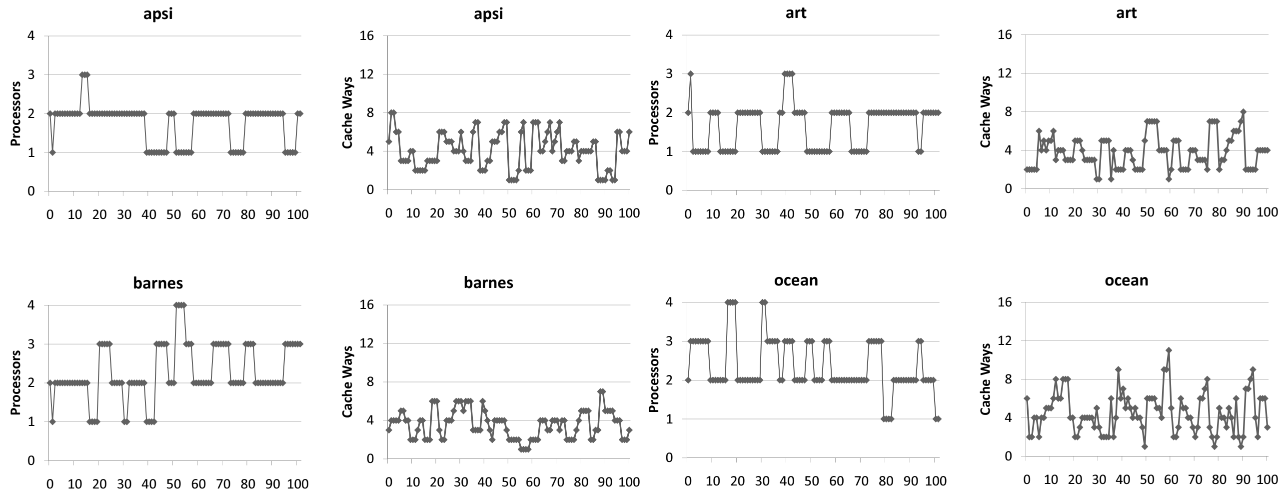


Figure 3: **Split up of the dynamically selected processor and cache partitions for a multiprogrammed workload consisting of apsi, art, barnes and ocean by Algorithm 2 and Algorithm 3, respectively, observed over 1 Billion cycles at intervals of 10 Million cyces. The x-axis represents different intervals of time and the y-axis represents the allocated number of processors/cache ways to each application during that interval. Note that the sum of the number of processors allocated to the four applications is equal to 8 and sum of cache ways is 16 at any instant of time.**

titioning, processor partitioning always precedes cache partitioning. However, due to finer time granularity of enforcement of cache partitioning (smaller enforcement interval), we may have cache repartitioning happening multiple times before processor partitioning of the next iteration (as illustrated in Figure 3). That is, we may have different cache partitions enforced for the same processor partition (during the same iteration), depending on the ratio between enforcement interval for processor partitioning and the enforcement interval for cache partitioning. We have implemented our integrated processor-cache partitioning algorithm (Algorithm 1) as a user level module on the Solaris operating system. The details of our experimental platform are discussed in the following section.

4. EXPERIMENTAL SETUP

Base System Configuration. We evaluate our integrated processor-cache partitioning approach on an eight processor CMP with 4-way issue superscalar processors. The baseline configuration of the CMP is as shown in Table 1. We simulate the complete system using Simics full system simulator [20]. All our results are obtained with the baseline configuration system described below.

Platform. We implemented our partitioning algorithm as a user level module on Solaris 10. We use the *pset_create()* and *pset_assign()* system calls in Solaris to dynamically create processor sets and assign processors to processor sets (effectively creating processor partitions) and the system call *pset_bind()* to assign processes to these partitions. We have implemented Simics modules to collect processor performance information like IPC values. Cache partitioning is implemented using way partitioning and is enforced upon a special call to a Simics module.

Benchmarks. To quantitatively evaluate the effectiveness of our integrated processor-cache partitioning approach on CMPs, we used multiprogrammed workloads of multithreaded programs consisting of programs with diverse computational and memory access characteristics. We have used six programs from the SPEComp 2001 benchmark suite, two programs from the SPLASH2 benchmark suite, and SPEC JBB, a commercial server benchmark. All the SPEC benchmark programs use the reference input set and are fast-forwarded to the beginning of their main loops. We run workloads (1) where all applications have high degree of parallelism and equally high demands for cache space. This scenario will benefit less from our scheme; (2) where applications have complimenting resource requirements and high degrees of variability in their resource requirements through the program execution. This scenario will have best results for our scheme; and (3) We chose the rest of the combinations in between the above two extremes. The workload mixes are shown in Table 2. As the number of threads in each application is crucial to determining the performance of our partitioning approach, it is important to know the number of threads in each application. All the SPEComp applications execute with eight threads each, while the SPLASH2 applications execute with sixteen threads each. SPEC JBB is configured with 16 warehouses and one client per warehouse. We warm up caches for approximately 1 billion cycles and collect statistics until the end of the loop iterations. The benefits obtained by using our partitioning approach are expected to be maximized when there is high variability in the scalability rates of the applications with respect to processors (in terms of processor affinity and/or degree of thread level parallelism) or cache space.

5. EXPERIMENTAL RESULTS

Schemes for Comparison. We study the performance of our proposed approach, *IntegratedPart*, in comparison with five other schemes. The first scheme, *EqualShare* that also forms our base case, is a static scheme that partitions both processors and L2 cache space equally among all the

Processors	8 processors with private L1 data and instruction caches
Processor Model	4-way issue superscalar
Private L1 D–Caches	Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency
Private L1 I–Caches	Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency
Shared L2 Cache	16–way set associative, 16MB, 64 bytes block size, 15 cycle access latency
Memory	4GB, 200 cycle off–chip access latency
QoS Threshold QoS_{Th}	-0.05 (-5%)
Enforcement Interval	50 Million cycles for processor partitions and 10 Million cycles for cache partitions

Table 1: Baseline configuration.

Mix	Applications
Mix 1	*apsi, art, barnes, ocean*
Mix 2	*apsi, SPEC JBB, mgrid, art*
Mix 3	*art, applu, ammp, apsi*
Mix 4	*mgrid, SPEC JBB , applu, ammp*
Mix 5	*mgrid, equake, SPEC JBB , barnes*
Mix 6	*equake, SPEC JBB , ammp, ocean*

Table 2: Various mixes of multithreaded applications considered for our multiprogrammed workload.

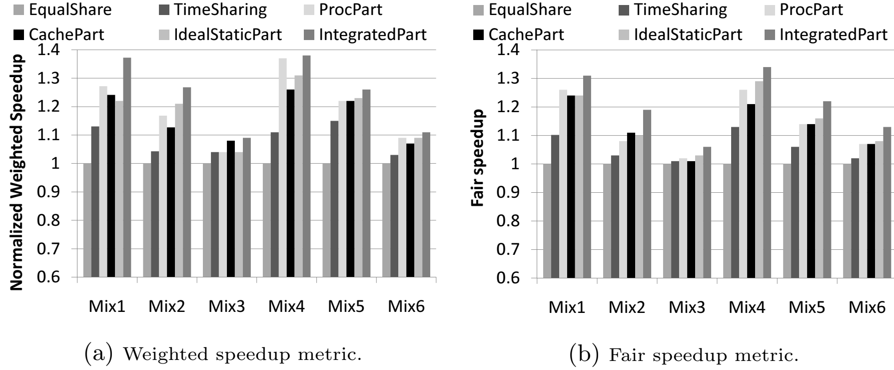
constituent applications of the multiprogrammed workload. The second scheme, *TimeSharing*, is the implicit processor partitioning (devoid of QoS) enforced by the underlying operating system scheduler with equal priority to all the constituent applications. The third scheme, *ProcPart* is our scheme with only processors being partitioned. The TimeSharing and ProcPart schemes do not impose any cache partitioning. As a result, the last level cache in these schemes is shared. The fourth scheme, *CachePart* is a state-of-the-art operating system directed cache partitioning scheme proposed in [27] (also similar to [17]). In particular, the cache partitioning policy implemented is the reactive miss rate equalization scheme of [27]. Note that this scheme represents the decoupled partitioning case, where the processors are implicitly partitioned by the underlying operating system and cache is also partitioned by the OS using reactive miss rate equalization scheme. ProcPart and CachePart are dynamic partitioning schemes for processors and cache respectively. The fifth and the final scheme, *IdealStaticPart* is the ideal static partitioning scheme that chooses the static partition of both processors and cache that delivers the highest fair speedup metric for the set QoS threshold value. We derived this by exhaustive simulation of a very large number of static allocations to each application (from 1 to 5 processors in steps of 1 processor, assuming that each of the four applications get at least one processor and from 1 MB to 13MB L2 cache in steps of 1 MB, again since each application gets at least 1 way of the 16 way associative cache or 1MB). The results shown in Figure 2 are a subset of these results. We further used this data to obtain the best static partition by computing the static partition of processors and cache that leads to the highest fair speedup metric. Note that this provides an upper bound on the best achievable performance using any static partitioning scheme. These five schemes, in our opinion, represent a large spectrum of available processor and cache partitioning techniques.

Comparison of Weighted Speedup Metric. Figure 4(a) plots the normalized weighted speedup metric of various schemes with respect to the EqualShare partitioning scheme. We observe that in all cases, the default TimeSharing scheme performs better than the base EqualShare case.

The EqualShare partition does not perform as well as TimeSharing as it statically enforces equal sharing of the processors across applications with varying demands, while the TimeSharing scheme distributes the processors more dynamically on demand basis as equally as possible among the applications.

The ProcPart scheme, on the other hand, dynamically adapts to varying demands and partitions the processors of a CMP with the objective of optimizing the fair speedup. Therefore, it fares better than the TimeSharing scheme. One of the important properties of multiprogrammed workloads that our partitioner exploits is the variability in the degree of thread level parallelism among the constituent applications of the workload. Therefore, as seen from Figure 4(a), the least benefits from ProcPart are obtained in case of Mix3, which is constituted of only SPEComp benchmarks that are having 8 threads each. In most cases (other than mix3), ProcPart performs better than cache partitioning alone. In mix3, cache partitioning alone performs better due to a disproportionately high gain in IPC for a single application (ammp). Note that such disproportionately high gains in one application do not reflect as an increase in the fair speedup metric (as seen from Figure 4(b) that plots the improvements in fair speedup metric). Another important observation from Figure 4(a) is that IdealStaticPart (the ideal static partition) does not even perform as well as processor partitioning or cache partitioning alone in some cases. Therefore, it is important to adopt a dynamic integrated processor cache partitioning in order to adapt to changing behaviors of applications. On an average, our IntegratedPart scheme performs, 24.67%, 15.02%, 4.47%, 6.89%, and 5.35% better in weighted speedup over the EqualShare, TimeSharing, ProcPart, CachePart and Ideal Static Part schemes, respectively.

Comparison of Fair Speedup Metric. Recall that optimizing the fair speedup metric is the primary objective of our partitioning scheme. Hence, we plot the fair speedup metric of the schemes based on optimizing the FS metric (ProcPart and IntegratedPart) in Figure 4(b). We can see that the perceivable improvements (magnitude of improvement) in the fair speedup metric are not as much as those with the normalized weighted speedup metric. The fairness metric is a combined measure of both total performance speedup and fairness among constituent applications of the multiprogrammed workloads. However, a relative comparison between the schemes is more appropriate and such a relative comparison shows that the integrated processor-cache partitioning approach fares better than both the processor-only partitioning scheme and cache only partitioning scheme in all cases on the fair speedup metric. Note that, improvements in the fair speedup metric does not necessarily correspond to a fair share of resources. Rather, it corresponds to fairness in the speedups achieved by applications over the equal share case. We would like to note that in our integrated processor cache partitioning scheme that maximizes fair speedup there were instances when one of the four applications (specifically, SPEC JBB in Mix-4) was allocated five processors (62.5% of available processors) and 2 MB of L2 cache (12.5% of available cache capacity) and at a subsequent point in time, was allocated one processor (12.5% of available processors) and 8 MB of L2 cache (50% of avail-



(a) Weighted speedup metric.

(b) Fair speedup metric.

Figure 4: Experimental results on an 8 core system with 16 MB, 16 way associative L2 cache.

able cache). It is important to note that such seemingly unfair resource allocations occur dynamically over different intervals of time so as to achieve fairness in speedups in presence of varying resource demands. On an average, our IntegratedPart schemes performs, 20.83%, 14.14%, 6.15%, 6.93%, and 5.07% better on the fair speedup metric over the EqualShare, TimeSharing, ProcPart, CachePart and IdealStaticPart schemes, respectively.

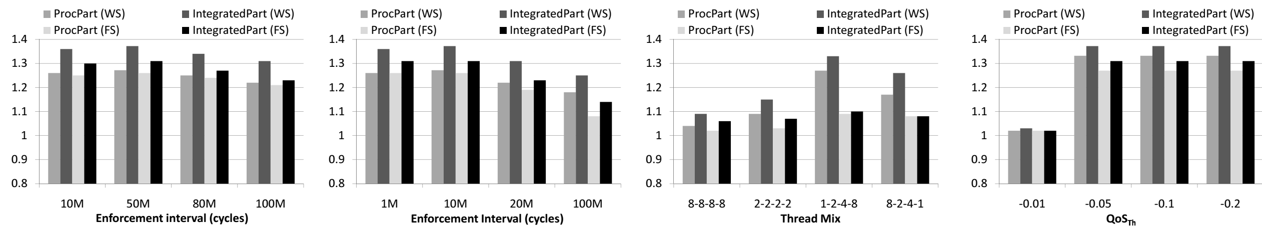
We also compared the performance of our integrated processor cache partitioning approach on a 16 core CMP with 32 MB L2 cache with processor only and cache only partitioning schemes. We observe that the integrated partitioning scheme achieves an average improvement of 9.79% over processor partitioning only scheme and 12.76% over cache partitioning only scheme with this larger CMP configuration. Also, it achieves average improvements of 8.21% and 9.19% over processor partitioning and cache partitioning schemes, respectively on the weighted speedup metric. Further discussion on the scalability of our integrated processor cache partitioning approach can be found in Section 6.

5.1 Sensitivity Analysis

Impact of varying EnforcementInterval. Recall that both ProcPart and IntegratedPart enforce each dynamically selected local optimal partition for a duration of EnforcementInterval. This interval parameter decides the frequency at which partitions are reconfigured. We studied the impact of varying this parameter from its default value of 10 Million cycles used for cache partitioning and 50 Million cycles used for processor partitioning in all our results in Section 5. The results of varying the EnforcementInterval for processor partitioning of a representative mix (Mix1) are presented in Figure 5(a). It is observed that we obtain better improvements as we reduce the reconfiguration interval from 100 Million to 50 Million cycles, as seen from Figure 5(a), as this provides a finer grain control on partitioning. However, further decreasing the interval (to 10 Million cycles) does not lead to additional benefits due to overheads involved in repeated reconfigurations. In a similar manner, Figure 5(b) plots results of varying the EnforcementInterval for cache partitioning of the same mix (Mix1) in the range of 1 Million - 100 Million cycles. It is seen that the best trade-offs between repartitioning overheads and performance improvements are achieved at 10 Million cycles interval (the performance with EnforcementInterval of 10M cycles is better than that with 1M cycles by about 1.5% although not clearly seen in the graph).

Impact of varying the number of threads per application in the workload. Variability in thread level parallelism is one of the key properties of workloads that our processor partitioner exploits. Therefore, we performed additional experiments by varying the number of threads per application for Mix3, the workload mix that had the least benefits from our approach. Figure 5(c) plots the normalized weighted speedup metric and the fair speedup metric obtained by varying the number of threads per application of Mix3. The default experiments were run with each constituent application of Mix3 being parallelized to have 8 threads. We see from Figure 5(c) that, the cases where there are variations in the number of threads clearly outperform the cases where the number of threads are equal between applications in terms of both the weighted speedup metric and the fair speedup metric. Another subtle, but important result to be noticed from Figure 5(c) is that the gains from a mix with 1, 2, 4, and 8 threads respectively of art, applu, ammp, and apsi is different from that with 8, 4, 2 and 1. The results vary based on CPU intensiveness of the applications with higher number of threads.

Impact of varying the QoS threshold. The QoS_{Th} parameter is a crucial parameter as it not only determines the level of performance guarantee, but also determines the extent of search space pruning that is possible. We evaluated our algorithm with various values of this metric for Mix1, which was the mix with standard deviation in the QoS metric closest to the average across all the mixes and static configurations we evaluated. The results are plotted in Figure 5(d). We see from this figure that, while having the QoS_{Th} at -1% reduces all the improvements, decreasing it beyond -5% (*i*.e., -0.1 and -0.2) does not earn additional benefits. This is primarily because of two reasons. We do not see high benefits with QoS_{Th} of -1% because there are hardly any partitions (other than the EqualShare itself) having QoS metric within -1% of the EqualShare case. Also, we do not earn additional benefits by decreasing QoS_{Th} beyond -5% because of the correlation between FS and QoS metrics and even in cases where $QoS_{Th} \leq -0.05$, each time, the partition that leads to QoS of within -5% also maximizes FS. Note that, if our objective metric was not FS (instead, was WS), better improvements could have been achieved in terms of WS by raising the bar for QoS limit. But, this could only be achieved by unfairly favoring one application over the other. To summarize, our approach also guarantees resilience to small errors in selection of the QoS_{Th} parameter and provides very good performance isolation.



(a) Sensitivity to EnforcementInterval of processor partitioning for a representative mix, Mix1 (apsi, art, barnes, and ocean).

(b) Sensitivity to EnforcementInterval of cache partitioning for a representative mix, Mix1 (apsi, art, barnes, and ocean).

(c) Sensitivity to variability in thread level parallelism for Mix3 (art, applu, ammp, and apsi).

(d) Sensitivity to Qos_{Th} of a representative mix, Mix1 (apsi, art, barnes, and ocean).

Figure 5: Sensitivity analysis. In these plots, WS and FS refer to weighted speedup and fair speedup respectively.

6. SCALABILITY

The ability of integrated processor-cache partitioning approach to scale to higher number of cores and larger caches (higher associativity for a way partitioned cache) is critical for its success and needs to be evaluated. Note that, search space pruning is highly scalable and can easily eliminate a very high number of configurations as the size of possible processor/cache shares to an application increases. As seen from Eq. 6, QoS_{Th} must be lesser than any partial sum of $\psi(i)$'s, but given the same QoS_{Th} with a larger CMP (with higher number of cores/cache ways), the number of partial sums of $\psi(i)$'s that are greater than QoS_{Th} increases exponentially. This leads to a large number of configurations being eliminated from consideration for evaluation in the process of finding the optimal configuration in the iterative partitioning approach.

We evaluated the performance of our integrated processor cache partitioning approach with a 16 core machine with 32 MB, 32 way associative shared L2 cache against that obtained on our default configuration of 8 cores with 16 MB, 16 way associative cache. The results with respect to fair speedup metric and weighted speedup metric are plotted in Figure 6(a) and Figure 6(b), respectively. Note that the fair speedup and weighted speedup metrics are computed with equal partitioning on the respective systems (8 cores, 16 ways equally shared in the 8 core case and 16 cores, 32 ways equally shared in the 16 core case).

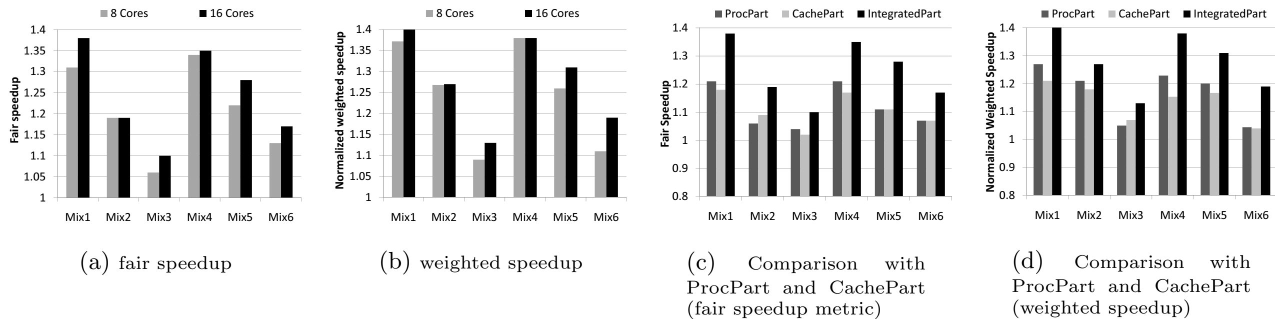
We observe that integrated partitioning approach performs better with respect to the equal share case on a 16 core CMP than that achieved on an 8 core CMP. On an average, in the 16 core case we see an improvement of 24.5% on the fair speedup metric over the equal share case as opposed to a 20.83% improvement obtained in the 8 core case. We also observed an improvement of 28.16% on the weighted speedup metric over the equal share case as opposed to a 24.67% improvement obtained in the 8 core case.

Another interesting property of the iterative partitioning approach is that it continues to provide resilience against large skews in either processor or cache partitioning alone when the number of cores or the number of cache ways scales to higher numbers. It does so by compensating the skew in one resource with an appropriate amount of the other resource as the integrated scheme computes the partitioning of cache in a manner that is aware of both the current processor partitioning as well as the overall goal of maximizing the fair speedup. Similarly, the processor partitioning in any iteration is aware of the cache partitioning in the previous iteration as well as the overall goal of fair speedup maximization.

We also compared the performance of our integrated processor cache partitioning approach on a 16 core CMP with 32 MB L2 cache with processor only and cache only partitioning schemes. The results of comparison on the fair speedup metric and weighted speedup metrics are plotted in Figure 6(c) and Figure 6(d), respectively. We observe that the integrated partitioning scheme achieves an average improvement of 9.79% over processor partitioning only and 12.76% over cache partitioning only scheme. Also, it achieves average improvements of 8.21%and 9.19% over processor partitioning and cache partitioning schemes, respectively, on the weighted speedup metric. These improvements are significantly greater than those obtained on an 8 core CMP with 16 MB L2 cache confirming that integrated partitioning performs better as we move towards larger CMPs with higher number of cores and more last level on-chip cache.

7. RELATED WORK

Recently, many researchers have explored CMP cache partitioning designs that attempt to alleviate inter-thread conflicts at the shared last level cache [27, 32, 17, 26, 6, 14, 12, 13]. Rafique [27] proposed an OS scheme that consists of a hardware cache quota management mechanism, an OS interface and a set of OS level quota orchestration policies for greater flexibility in policies. Our approach to enforcement of cache partitions is similar to this work. Chang and Sohi [6] have proposed cooperative cache partitioning, wherein they use multiple time sharing partitions to resolve cache contention. They use similar metrics as ours (fair speedup and QoS metric) to quantify the benefits from their approach. Qureshi and Patt [26] proposed a utility based cache partitioning scheme where the share received by an application was proportional to the utility rather than its demand. Our approach can be viewed as a utility based approach, where the relative utility of processors/cache is directly measured based on the feedback from performance counters. Since our algorithm is implemented in software, we have greater flexibility and scope to search for the best processor-cache partition. Apart from these throughput oriented strategies, ensuring QoS in shared cache partitioning has also been widely discussed in the literature [14, 12, 13].



(a) fair speedup

(b) weighted speedup

(c) Comparison with ProcPart and CachePart (fair speedup metric)

(d) Comparison with ProcPart and CachePart (weighted speedup)

Figure 6: Scalability results on a 16 core system with 32 MB, 32 way associative L2 cache.

All these studies are oblivious to processor partitioning. In fact, these studies are counterparts of the cache partitioning strategy that we adopt. It is important to note that, our approach is flexible enough in the sense that, any of these cache partitioning techniques can be combined with our processor partitioning approach. We use a uniform, regression analysis based strategy to partition both processors and the L2 cache space in order to show the versatility of our approach to prune the search space in either cases. This does not, however, restrict the usage of any hardware-based cache partitioning mechanism along with our OS directed processor partitioning. The key take-away here is the need to perform integrated processor-cache partitioning.

Yue and Lilja [36] demonstrated the feasibility and studied the performance benefits of dynamic processor partitioning on multiprocessor system. Apart from works on explicit partitioning, several classical scheduling techniques [16, 34] implicitly partition the processors. Most of these scheduling techniques perform well with the objective of increasing the utilization, but are not well suited for optimizing the throughput and providing performance isolation for high utilization systems. Recently, Bower et al [3] studied the challenges involved in design of schedulers for dynamically heterogeneous multicore processors. We would also like to note that we consider homogeneous multicore processors in this study as they represent the vast majority of existing processors. However it is interesting to extend our study to heterogenous multicore processors [18, 1] or asymmetric single ISA multicore processors [21] and is part of our future work. Almost all these scheduling schemes are oblivious to cache allocation as caches are typically hardware managed and most schedulers do not utilize feedback from the hardware about the impact of scheduling on cache management, while we propose a hardware and software co-design based partitioning scheme to ensure better system performance.

There has been some recent work on shared cache aware schedulers. Tam et al [33] proposed a thread clustering scheme to schedule threads based on detecting sharing patterns online by using the data sampling features of the performance monitoring unit. Our approach differs from this work in determining suitable processor partitions and adapting to the changing levels of application parallelism. The clustering of threads in our case is implicit due to partitioning of processors on an application basis. Fedorova et al [8] proposed an algorithm called cache-fair that reduces co-runner-dependent variability in an application's performance by ensuring that the application always runs as quickly as it would under fair cache allocation, regardless of how the cache is actually allocated. Although this scheme provides performance isolation and reduces variability in performance, it does not take advantage of knowing the partitioning to improve the performance. It is oblivious to the cache partitioning and emulates a fair cache partitioning by suitably modifying the time slices allotted to the application processes.

8. CONCLUSIONS AND FUTURE WORK

Integrated processor-cache partitioning proposed in this paper achieves significant improvements over partitioning of either the processors or the shared L2 cache in isolation. We proposed a regression based prediction model to predict the behavior of applications and techniques to prune the large search space in order to find the most suitable processor and cache partitions. Extensive simulations using a full system simulator and a set of diverse multiprogrammed workloads show that our integrated processor-cache partitioning approach performs, on an average, 20.83% and 14.14% better than equal partitioning and the implicit partitioning enforced by the underlying operating system, respectively, on the fair speedup metric on an 8-core CMP system. We also compare our approach to processor partitioning alone and a state-of-the-art cache partitioning scheme and our scheme fares 6.15% and 6.93% better than these schemes. Additionally, as compared to the ideal static partitioning of both processors and the cache our approach brings an improvement of 5.07% on average on the fair speedup metric across a set of diverse multiprogrammed workloads on an 8 core system. As part of our future plans, we will extend this work to solutions for virtualization workloads with multiple guest operating systems.

9. REFERENCES

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Comput. Archit. News*, 33(2), 2005.

[2] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proc. of the International Symposium on Microarchitecture*, 2008.

[3] F. Bower, D. Sorin, and L. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28(3), 2008.

[4] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. of the International Symposium on High-Performance Computer Architecture*, 2005.

[6] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. of the International Conference on Supercomputing*, 2007.

[7] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3), 2008.

[8] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. of the International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[9] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. Chip multithreading systems need a new operating system scheduler. In *Proc. of the SIGOPS European Workshop*, 2004.

[10] R. Gabor, S. Weiss, and A. Mendelson. Fairness enforcement in switch on event multithreading. *ACM Trans. Archit. Code Optim.*, 4(3), 2007.

[11] F. Guo, H. Kannan, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis. From chaos to QoS: case studies in CMP resource management. *SIGARCH Comput. Archit. News*, 35(1), 2007.

[12] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[13] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proc. of the International Conference on Supercomputing*, 2004.

[14] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. *SIGMETRICS Perform. Eval. Rev.*, 35(1), 2007.

[15] L. K. John. More on finding a single number to indicate overall performance of a benchmark suite. *SIGARCH Comput. Archit. News*, 32(1), 2004.

[16] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1), 1988.

[17] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[18] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11), 2005.

[19] K. Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in smt processors. In *Proc. of the International Symposium on Performance Analysis of Systems and Software.*, 2001.

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2), 2002.

[21] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 28(3), 2008.

[22] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. of the International Symposium on Microarchitecture*, 2007.

[23] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proc. of the International Symposium on Microarchitecture*, 2006.

[24] K. J. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore resource management. *IEEE Micro*, 28(3), 2008.

[25] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. *SIGOPS Oper. Syst. Rev.*, 41(3), 2007.

[26] 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 the International Symposium on Microarchitecture*, 2006.

[27] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[28] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *Proc. of the International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[29] A. Sen and M. Srivastava. *Regression Analysis*. Springer, 1990.

[30] J. E. Smith. Characterizing computer performance with a single number. *Commun. ACM*, 31(10), 1988.

[31] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[32] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1), 2004.

[33] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proc. of the European Conference on Computer Systems*, 2007.

[34] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proc. of the USENIX conference on Operating Systems Design and Implementation*, 1994.

[35] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. In *Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2005.

[36] K. Yue and D. Lilja. Dynamic processor allocation with the Solaris operating system. In *Proc. of the International Parallel Processing Symposium*, 1998.