The Pennsylvania State University
The Graduate School

A WORKLOAD MAPPING METHOD FOR MULTICORE SYSTEMS USING CROSS-RUN STATISTICS

A Thesis in
Computer Science and Engineering
by
Mahmut Sami Aktasoglu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2012

Abstract

Multicore architectures have become the design of choice in today's microprocessor market. Due to the significant strides made in process technology and poor scalability of the monolithic design, multicore processors are used in more and more devices nowadays. Multicore processors provide applications with a variety of resources in a single chip such as caches, memory controllers, etc. It is not a trivial task, however, to utilize such resources among applications efficiently due to dynamic and unpredictable nature of destructive interference among applications. Multicore systems and operating systems (OS) offer a limited number of tools to manage on-chip resources. Some hardware solutions existing in today's multicore architectures are Dynamic Voltage/Frequency Scaling (DVFS) and capability of switching the cores on/off. Software solutions offered by operating systems are limited to use underlying architecture's methods and scheduling. Scheduling in multicore systems is a means of providing applications and threads access to system resources. OS schedulers not only decides when to dispatch application threads, but also maps them to specific cores which defines the resources available for the thread. Therefore, application mapping is a key method for mapping resources to applications on systems level.

To better utilize resources where multiple application threads are running, resource-application mapping problem has to be addressed. In this work, we address the problem by presenting a resource-application mapping method using cross-run statistics. Our work is unique in the sense that our method handles various architectures and workload sizes while exploiting the data collected from prior executions of the applications, i.e., cross-run statistics. We present an algorithm which uses these statistics to decide the mapping between applications and system resources to improve overall performance. Our results collected on commercial machines show that our scheme can improve overall system performance by up to 20% over the default OS scheduler.

Table of Contents

List of Figures **v**

List of Tables **vi**

Chapter 1
Introduction **1**

Chapter 2
Background **4**

Chapter 3
Application Mapping Using Cross-Run Statistics **6**
3.1 Motivation 6
3.2 Definitions 11
3.3 Mapping Formulation 12
3.4 Algorithm 14
3.5 Example 16

Chapter 4
Experimentation **18**
4.1 Experimental Setup and Methodology 18
4.2 Experimental Results 23
4.3 Observations and Discussions 29

Chapter 5
Conclusion and Further Discussions **33**

Bibliography **38**

List of Figures

3.1 A schematic view of CMPs used: Intel Harpertown [1] and AMD Magny-Cours [2] processors. 7
3.2 Performance of two mappings for two workloads that have common applications. Higher values are better. 8

4.1 Performance of workload 6C0 over consecutive mapping intervals. . 23
4.2 Performance of workload 8C0 over consecutive mapping intervals. . 24
4.3 Performance for 8-application workloads. Error bars show maximum and average case performance at the last mapping interval. *Avg.* is average of workloads with at most 2 non-familiar application. *All Avg.* is average of all workloads. 26
4.4 WSU of each possible mapping for the 8-application workloads at the last mapping interval. 27
4.5 $\sum$MIpS of each possible static mapping for the 8-application workloads at the last mapping interval. 28
4.6 WSU of each possible static mapping for the 12-application workloads at the last mapping interval. 29
4.7 $\sum$MIpS of each possible static mapping for the 12-application workloads at the last mapping interval. 30

List of Tables

3.1 Sample history records of a 4-application workload. The values are pair-wise sum of speedups of the applications in the corresponding rows and columns. 16

4.1 Major system configuration and experimental parameters. 19
4.2 Application mixes for 6-application workloads. The non-familiar applications are bolded. 20
4.3 Application mixes for 8-application workloads. The non-familiar applications are bolded. 20
4.4 Application mixes for 12-application workloads. The non-familiar applications are bolded. 21
4.5 Randomization thresholds (i.e., average number of records per application required) for various sizes of familiar applications with a fixed number of records. 25
4.6 Difference in scores of application pairs with different input set size. Applications on the top row have at least two input sets provided. They are co-run with applications on the leftmost column for each input set they have. Results presented are percent difference of pairwise performance with different input sets. 31

Chapter 1

Introduction

Multicores are the architecture of choice due to significant strides in process technology and the poor scalability of monolithic designs in power and complexity. Today's microprocessors offer more computational power packed in a single chip. They provide a set of resources such as cores, caches, memory controllers and even graphics processors on the very same chip. This implies better performance for most applications as well as running more applications and threads concurrently. However it is not a trivial task to take advantage of the multicore machines to the fullest. This problem is difficult due to sharing of *on-chip resources*. On-chip resources are limited and applications/threads contend over them. This implies *destructive interferences* [3, 4, 5] across concurrently-running applications/threads. An example of the destructive interferences is displacement of on-chip data of an application due to an access made by another application. The key to achieving best performance from multicores is to address the contention problem via resource management.

There are several ways to address resource management problem on a Chip Multiprocessor (CMP). The first and most trivial way is to identify possible contention sources and improve the program code. The program could be tailored very carefully by the programmer to exploit the features and resources of the multiprocessor (as well as other off-chip resources). This is a tedious task if left to the programmer alone since it is very hard (if not impossible) to come up with a piece of code that runs well on all types of hardware configurations. At this point, compiler is the programmer's best friend to help mitigate the optimization of the code for different hardware and extract more performance. Many compilers offer a variety of automatic parallelization or resource-specific optimization features. However, compilers are oblivious of dynamic nature of a program that can be acquired only during run-time (such as memory access patterns, program parameters and inputs). Moreover, neither compilers nor programmers are capable of predicting the effects of destructive interferences of other applications/threads accessing the same resources.

Yet another way of resource management is to use run-time systems. Run-time resource management can adapt to changing requirements of the applications and the system. However, they usually come with an overhead that needs to be mitigated by the improvement provided. This overhead comes in forms of monitoring application characteristics off-line (i.e., before application is run) or on-line (periodically during run time), and distributing the resources. Run-time systems need efficient ways of monitoring. A recent trend in multiprocessor architecture is to provide a dedicated hardware, namely Performance Monitoring Units (PMUs), to measure important statistics about the system and applications. PMUs enable run-time systems to gather crucial micro-architectural information such as cache miss rates, number of instructions executed, etc. Even though run-time systems have an efficient method to gather required data, they still lack efficient ways of distributing the resources. Prior studies try to tackle the problem of resource distribution with various strategies. Most of the proposed solutions assume a specific hardware unit to achieve efficient resource distribution [6, 7, 8, 9, 10]. Others simply rely on mapping applications on specific cores to minimize shared resource contention [3, 11, 12, 13, 14, 15, 16, 17, 18]. Unfortunately, many promising hardware solutions (i.e., cache practitioners) have not been realized in commercial products. Therefore, mapping applications to cores is the only way to go to implement a run-time system in today's multicore architectures.

In this work, our goal is to realize a run-time system for resource management on CMP systems that is capable of handling a variety of resources and architectures with low overhead and no additional hardware requirement. Our proposal, different from aforementioned studies, does not assume any additional hardware or software capabilities other than what is available on most of the current multicore systems. Moreover, our scheme differs from the rest by taking advantage of *cross-run statistics* (i.e., data collected from previous runs). We argue and show that cross-run statistics of applications hold valuable information that can be exploited to achieve better overall performance. **For this purpose, we try to quantify the benefits of exploiting cross-run statistics in improving the performance of future runs in the context of multicore architectures.** Specifically, we make the following **contributions**:

- We provide a simple mathematical analogy for resource mapping problem. We describe resources and mappings in terms of simple linear equations and constraints, which allows us to project the problem as an optimization of a single linear objective function. Our model enables us to generalize the scheme for any type of resources and workloads.
- We illustrate the benefits and uses of cross-run statistics. Moreover, we discuss practical implications and provide a comprehensive study on how such data can be used.
- We present a novel strategy that uses cross-run statistics to decide good workload mappings in future runs where multiple applications compete for on-chip resources. To our knowledge, this work presents the first approach to the resource contention problem in multicores that exploits statistics collected over previous invocations of applications.
- To test the effectiveness of our approach, we measure weighted speedup (WSU) [19] and raw throughput (i.e., million instructions executed per second, MIpS) of two systems. The results show that our approach is able to use the information from previous runs (i.e., history) to provide 11% and 8% improvement over the average case for a variety of workloads of different sizes for WSU and MIpS, respectively.

In the next chapter, we present a brief background on resource management research. In Chapter 3, we discuss the details of our resource management method. We evaluate our method and illustrate the experimentation results in Chapter 4. Finally, we conclude our thesis in Chapter 5 and discuss limitations and further research possibilities.

Chapter 2

Background

Resource management on multicore systems is a well-studied problem. Previous studies [3, 4, 5, 18, 20] discussed inter-application interference. Performance models and prediction techniques have been proposed to estimate the degree of degradation resulting from the contention on different resources [17, 18, 20, 21, 22, 23].

Prior studies address various on-chip shared resources. There are studies that focus on contention caused by processors, memory bus, prefetchers, memory controllers and I/O bus in multicore architectures [12, 21, 22, 24]. Among shared resources, contention on last-level caches has been explored extensively. In earlier studies, cache contention is mainly mitigated in two ways: Cache partitioning [6, 7, 8, 9, 10], and thread scheduling [3, 12, 13, 14, 15, 16, 17, 18].

Knauerhase et al. [16] propose several scheduling policies for balancing last-level cache misses. Kazempour et al. discuss effects of caches on various multiprocessor types in [18]. Fedorova et al. develop a scheduling algorithm to reduce unfair cache usage in [13]. They further propose a cache-aware scheduling algorithm in [25] that co-schedules threads to achieve low L2 cache miss rates and gives priority to threads that do not require much space in the L2 caches. In [14], Fedorova et al. propose a thread scheduling policy based on a reuse-distance model to estimate the cache miss ratio for each workload. In another study, Fedorova et al. present cache-fair algorithms that offset the variability in the IPC by adjusting CPU timeslice based on heuristic cache models [15].

Chen et al. [3] compare the parallel depth-first and the work stealing algorithms for scheduling fine-grained multithreaded programs based on the constructive cache sharing among threads. Tam et al. [17] design a sharing-aware thread scheduling policy, which reduces the long latency of cross-chip cache accesses. Snavely and Tullsen [19] develop a symbiotic job scheduling method that co-schedules threads on SMT by sampling a large number of thread assignments and picking the ones with the best observed rate of instructions per cycle.

Recent work [21] investigates different classification schemes for threads, including stack distance competition [20], animal classes [26], solo miss rate [16] and the pain metric, and then design and evaluate two scheduling algorithms, named distributed intensity (DI) and distributed intensity online (DIO), based on the miss rate classification scheme and the centralized sort scheduling policy.

Bhadauria and McKee [12] propose a dynamic scheduling scheme for multithreaded and multiprogrammed workloads to optimize a global system metric (throughput per Watt).

Our work is different from these previous studies since we use cross-run statistics to find good application-to-core mappings. Our work does not assume prior knowledge about the applications to be mapped, nor do we require additional hardware. Mapping decisions are made both off-line (i.e., before the workload starts execution) and on-line (i.e., periodically throughout the execution). Our scheme can be used orthogonally with other dynamic mapping schemes proposed in the literature to find an initial mapping, as well. To the best of our knowledge, this work is the first that explores the use of history statistics from previous runs to devise good mappings in the future.

Chapter 3

Application Mapping Using Cross-Run Statistics

History based mapping relies on past information about prior application runs to map the application to system resources in future runs. We begin the presentation with our motivation.

3.1 Motivation

Prior studies mostly address contention problem for a limited set of resources. However, performance degradation is a combined result of contention on all resources. Addressing contention from the perspective of a single resource, however, might instead create new bottlenecks. This is because such an approach neglects the complex relation among various shared resources. Consider a scenario in which the major bottleneck is off-chip memory bandwidth. Improving contention only on this resource would inevitably improve performance of memory-bound applications. However, this might cause more contention on caches and processors since applications will avoid extra stalls and execute more instructions. The applications whose performance is increased will be more aggressive using other resources (e.g., caches and core quanta), adversely affecting other applications' performance.

When addressing resource contention, it is important to assess the tradeoff among different applications to achieve higher system performance. Conflicts of interest are likely to arise among concurrently-executing applications. Typically,

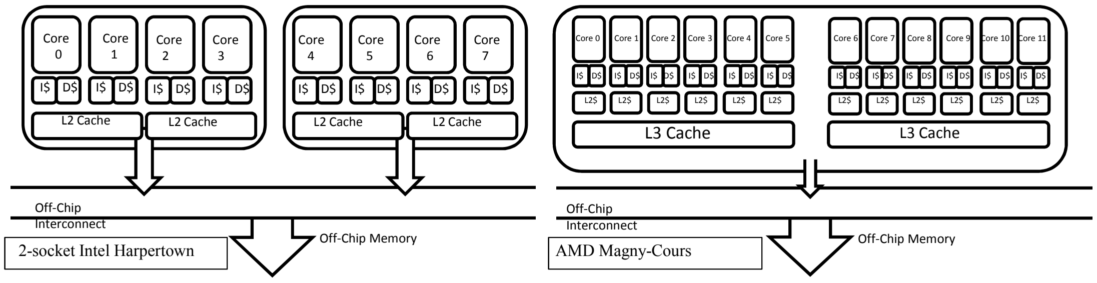
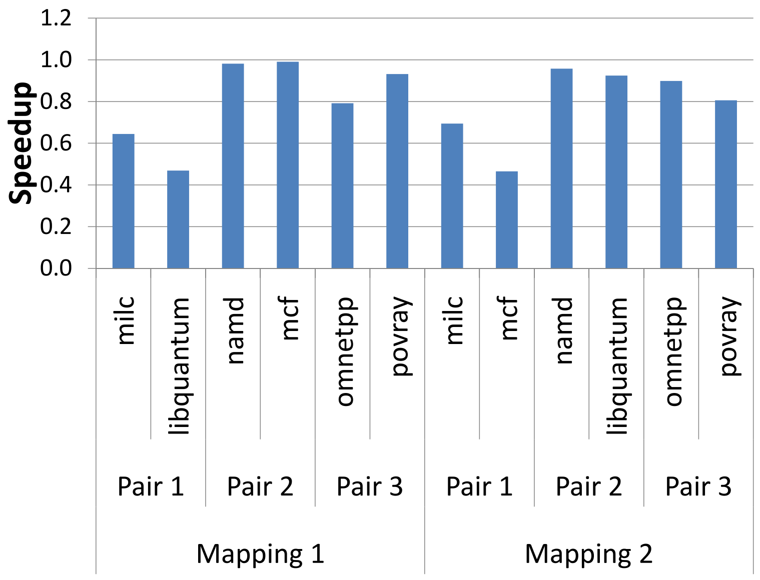


Figure 3.1: A schematic view of CMPs used: Intel Harpertown [1] and AMD Magny-Cours [2] processors.

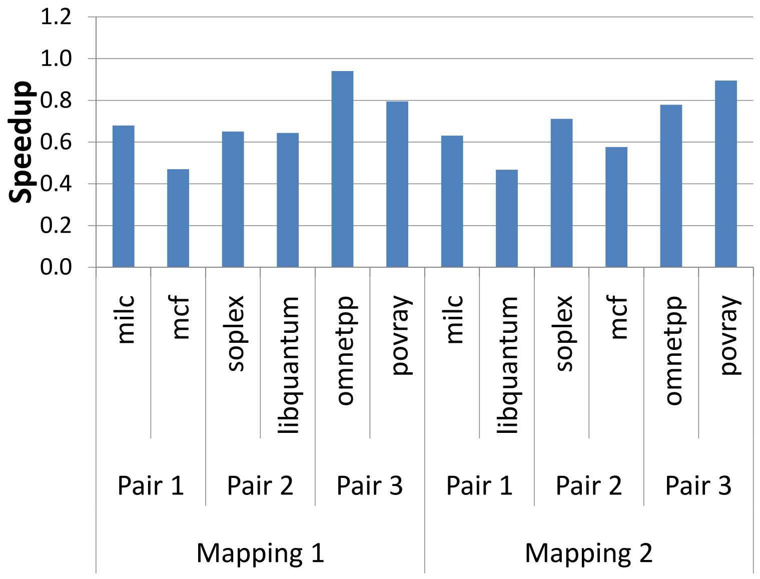
a decision will yield (different amounts of) improvement in some applications, whereas the rest will experience more performance degradation. Therefore, success of resource management schemes depends strongly on how they capture such tradeoffs. The complexity of interactions among shared resources and applications raises the need to tackle resource contention at a system level rather than considering only a single resource or application at a time to achieve better performance in the context of multi-application workloads.

To better understand resource contention, we closely investigated various workloads and mappings. In Figure 3.2, we present results for two sample workloads of 6 applications. Each workload is run for 20 seconds on 3 core pairs (sharing the last level cache) of a dual-socket Intel Harpertown machine (see Figure 3.1). In this scenario, applications mostly contend for dual-shared L2 caches. The results in Figures 3.2a and 3.2b illustrate two possible mappings for each workload. The first mapping (Mapping 1 in both figures) represents the optimal mapping for that workload. Each pair of applications is pinned to a core pair sharing an L2 cache. Note that two workloads differ by only one application (namd in Workload 1 vs. soplex in Workload 2). The vertical axis corresponds to speedup with respect to an application's solo performance.

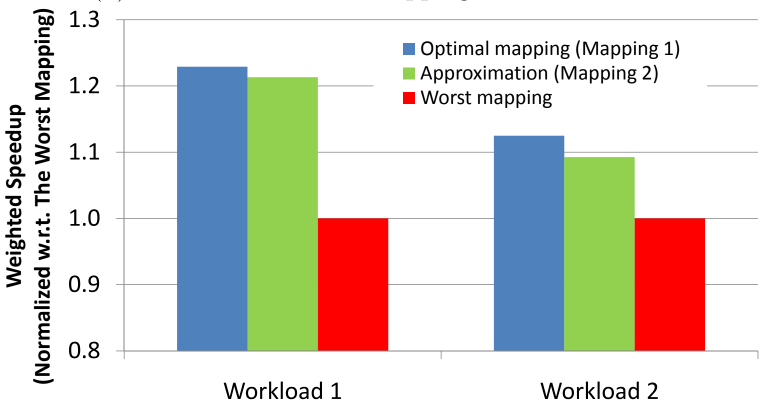
As shown in both Figures 3.2a and 3.2b, applications that run together with *soplex* have a lower performance compared to when they run with *namd* (see *mcf* in Mapping 1 in Figure 3.2a vs. Mapping 2 in Figure 3.2b). This suggests that *soplex* is more resource demanding than *namd*, and it hurts the performance of its co-runner. We also observe that replacing *namd* in Workload 1 with *soplex* causes degradation only to its co-running application. In other words, the other



(a) Performance of two mappings for Workload 1.



(b) Performance of two mappings for Workload 2.



(c) Overall performance of different mappings.

Figure 3.2: Performance of two mappings for two workloads that have common applications. Higher values are better.

application pairs' performance are not affected much when we replace an application with a more resource demanding one. This is because *soplex* contends on the partially-shared last-level cache and the system we use has a symmetric (dual-shared) cache structure. As a result, there is little interaction with applications on other cores, causing little or no extra cache contention. Each application pair is actually executing as if it was isolated from other pairs.

This observation holds for most application groups we tried. It is important as it enables us to predict the performance of mappings by using information from other mappings of the same workload, or from other workloads that have applications in common. If we can define shared resource groups such that the interactions among groups are kept at minimum, then we can expect that this observation will hold. In our current example, the application pairs share cache on level 2, which is the main contention point. Therefore, we choose to form shared resource groups based on level 2 cache contention.

Next, we demonstrate how this information can be exploited. Recall that the mappings labeled as "Mapping 1" in Figures 3.2a and 3.2b represent optimal static mapping, which achieves the highest system performance among all possible static mappings. Let us assume that we executed Workload 1 using Mapping 1, and saved pair-wise performance data for the applications. Let us further assume that we know that this mapping is the optimal static mapping for this workload. Now, we need to find a mapping for Workload 2, which consists mostly of applications that were previously in Workload 1. If we have not executed the second workload before, there is no way to pro-actively determine a mapping that would reduce resource contention. However, since we know that pair-wise application performance is loosely related to other application pairs in the workload, we can use this information to determine a mapping for the new workload. Specifically, for Workload 2, we chose the same pairs as Workload 1's optimal mapping for the five applications that are in both workloads. The resulting mapping is Mapping 2 for Workload 2. Similarly for Workload 1, we choose the same pairs from optimal mapping of Workload 2 for applications that are common in both workloads. The resulting mapping is Mapping 2 of Workload 1.

For both the cases, Figure 3.2c shows the comparison of the optimal mappings (Mapping 1), the mappings formed from the other workload's history data (Mapping 2) and the worst case mappings for the two workloads. The vertical axis shows weighted speedup of each static mapping normalized to the worst static mapping. Weighted speedup [19] is calculated as $\sum_{i=0}^{n} IPC_i^{in\ Workload}/IPC_i^{Solo}$, where n is the of applications in a workload. This scenario reveals an important use of history data. By employing a simple heuristic, a near-optimal mapping can be found.The performance benefits over worst-case mappings are 21% and 9% in Workloads 1 and 2, respectively.

It is important to note that the information collected from a workload execution could be useful for other workloads as well. As a use case, consider clustered computer systems. In many cluster environments, resource management is handled by a job scheduler that distributes jobs to nodes based on the amount of resources demanded by users. This might cause different jobs from different users to be executed on the same node (i.e., time sharing), which is typically a multicore system. Resource management on the node-level is needed when several applications co-run on a single node and should be addressed through a static or dynamic scheme. Most of the previously proposed mapping and scheduling schemes [3, 13, 14, 15, 16, 17, 18, 27, 28] try to eliminate contention at a particular shared resource only. Moreover, these schemes do *not* make use of any off-line information about characteristics of the applications nor do they keep track of the frequently-executing workloads. Existing dynamic schemes typically monitor applications before adjusting mapping decisions. This incurs unnecessary runtime overhead for repeating workloads, or leaves some performance gains on the table that could otherwise be collected, if we were able to start with a better initial mapping.

Based on these experiments and observations, we advocate using history data from applications in multi-programmed workloads. History information is useful as it provides hints to determine good mappings without hardware or runtime overhead. For this purpose, we developed an algorithm that finds near-optimal static workload mappings. In the following sections, we present the definitions we use to create a formal problem statement.

3.2 Definitions

Shared Resource Group is a group of cores connected to a specific shared hardware unit on-chip. In this work, we limit each core to execute only one application. Therefore, we do not consider cores as shared resource units. The shared hardware unit can be last level cache, memory controller, on-chip interconnect, etc. We assume that applications executing on a shared resource group have insignificant or no impact on applications running on other shared resource groups. This implies each application in a group has equal share of private resources (e.g., L1 caches), and competes with other applications for the shared unit available only in its own group (e.g., last-level shared cache).
History Record is a record that holds a performance score for a set of applications that have executed together in a group.
System Topology is a representation of the system in terms of shared hardware units and resource groups. It identifies the cores connected to each shared hardware unit in the system. Each shared unit has a corresponding shared resource group. We represent resource sharing on a given topology by a matrix, which we call *Topology Matrix*. A system can be represented by a topology matrix (T), defined as follows:

$$T_{ij} = \begin{cases} 1, & \text{if } c_j \in G_i \\ 0, & otherwise, \end{cases}$$

where c_j is the j^{th} core and G_i is the i^{th} shared resource group. This matrix defines the core list for each shared resource group. For instance, the topology matrix of a 4-core Intel Harpertown processor for dual-shared last level (L2) caches (T_H):

$$T_H = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

This matrix tells (logical) cores 0&3 and 1&2 share L2 caches. Similarly, topology matrix for a dual-socket Harpertown configuration (T_{H2}) is below:

$$T_{H2} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Note that our topology matrices do not reflect physical layout of the processors, but define the topology of the logical processors as defined by the operating system. Each row corresponds to a shared resource group. There are as many columns as the number of cores in each configuration. Cores on the same row contend for the resources.

We have constraints for our topology matrix in each shared resource level that guarantee that each core belongs to a single shared resource group. The constraints are:

$$\sum_j T_{ij} = p, \ \ \forall i, i = 1 \ldots g \tag{3.1}$$

$$\sum_i T_{ij} = 1, \ \ \forall j, j = 1 \ldots c \tag{3.2}$$

where p is the number of processors in each resource group and g is the number of resource groups.

3.3 Mapping Formulation

We now introduce a representation that describes the mapping of applications to cores. For any multicore system, the resource management problem can be treated analogously to the application mapping problem at the user level. This is because, in modern systems, it is not possible to assign on-chip resources partially or completely to applications even at the OS level. In case of partially-shared resources, accesses to the resources solely depend on how the OS scheduler assigns the application to cores. As a result, the amount of contention among applications depends on the resources that are available to the cores on which they are running.

Based on system topology, resource contention can be managed by choosing the cores to which applications will be mapped at the user level. For this purpose, we define a matrix that we call the *Application Mapping Matrix*, A, as follows:

$$A_{ij} = \begin{cases} 1, & \text{if } App_i \text{ is mapped to } G_j \\ 0, & otherwise. \end{cases}$$

This matrix indicates which applications contend on which shared resources. We call the set of applications mapped to a certain shared resource group as *Workload Partition* (or simply *partition*), which is denoted as WP_i. We assume that a workload has w applications and the number of applications in each workload partition is the same in a homogeneous system.

$$\sum_j A_{ij} = 1, \ \ \forall i, i = 1 \ldots w \tag{3.3}$$

$$\sum_i A_{ij} = \frac{w}{g}, \ \ \forall j, j = 1 \ldots g \tag{3.4}$$

Constraints (3.3)-(3.4) enforce that an application belongs to only one partition.

As the final step, we introduce a formal representation for mapping applications to individual cores on the systems (i.e., applications' affinity). For this purpose, we define *Application Affinity Matrix, M,* as:

$$M_{ij} = \begin{cases} 1, & \text{if } App_i \text{ is allowed to run on } c_j \\ 0, & otherwise. \end{cases}$$

Using our Application Affinity Matrix, the mapping problem can be restated as:

For a workload of **w** *applications, on a system with c cores and g shared resource* *groups, find a* $\mathbf{w} \times \mathbf{c}$ *Application Affinity Matrix, M, with the following constraints:*

$$\sum_j M_{ij} = p, \ \ \forall i, i = 1 \ldots w \tag{3.5}$$

$$\sum_i M_{ij} = \frac{w}{g}, \ \ \forall j, j = 1 \ldots g \tag{3.6}$$

By definition, *Application Affinity Matrix M* is a mapping of w applications to $c = p\ g$ cores. Constraints (3.5)-(3.6) is intended to force applications to run on a subset of cores.

Finally, our goal (i.e., desired output) for application mapping problem is finding the right application affinity matrix M. This matrix can be represented using the underlying multicore topology and application mapping matrices as follows:

$$\mathbf{M} \ = \ \mathbf{A}\ \mathbf{T}. \tag{3.7}$$

3.4 Algorithm

We sketch our solution to find M in Algorithm 1. We define a *Mapping Interval* during which we monitor the applications. At the end of each mapping interval, we gather statistics, update history records and run Algorithm 1 to find a mapping for the next interval.

The algorithm starts with the initialization of topology and history data specific to the workload. Next, it defines *Adventure Threshold* and *Randomization Threshold* (Lines 2-3). After the initialization step, it finds the affinity matrix by either pseudo-randomly generating it, or by consulting the history data (Lines 4-8).

The mappings found by our algorithm depend on the history records. A new history record is saved for each application group running on the same shared resource group in an interval. History records are created or updated either at the

Algorithm 1 History-based Workload Mapping

```
Require: $w$ Workload Size
    $n$ Number of familiar applications so far
    $App_1 ... App_w$ Workload of $w$ applications
    $C_1 ... C_c$ List of $c$ cores
    $T$ Topology Matrix
    $Adv$ Adventure Threshold
    $Thr$ Randomization Threshold
    $History\ Records$ Data collected from previous executions
Ensure: Mapping $M$ that binds each $App_i$ to a core $C_k$ where $i = 1...w$ and
    $k = 1...c$
1: $Workload\ History \leftarrow App_i$ data from $History\ Records$
2: $Adv \leftarrow 0.1$ {Probability of mapping the workload randomly}
3: $Thr \leftarrow round$ ( 0.6 x $n$ )
4: if Average number of records < $Thr$ OR $random$ ( 0, 1.0 ) < $Adv$ then
5:     $Pseudo-randomly$ create $p$ workload partitions $WP$ where $p$ is the number
       of $Shared\ Resource\ Groups$
6: else
7:     $A \leftarrow \max_A \frac{1}{n} \sum_{i,j} A_{ij}\ score(A_{ij}, WP_j)$
8:     Return $M = A\ T$
```

end of mapping interval or after one of the applications terminates.

Initially when there is little or no history data, our algorithm pseudo-randomly chooses mappings to populate the history records. For this purpose, we group applications that have not co-run in the same group previously (hence 'pseudo-random'). As a result, the number of records for each application is increased until there are enough records as defined by a *randomization threshold.* The mapping that our algorithm determines for a given workload will most likely be the same unless new history records are added or existing records are updated. However, this may not be desirable if we want to find better mappings over time. An *adventure threshold* is introduced to obtain information about mappings that are not tried before, possibly revealing better solutions.

After the randomization threshold is reached, the algorithm seeks the mapping with the *highest score* among all possible mappings (Line 7). At Line 7, the total score over all possible workload partitions (WP) are maximized. The function *score* returns the value in the history record for partition WP_j that contains application A_{ij}. If there is no record of A_{ij} running with other applications in WP_i,

	App1	App2	App3
App2	0.967	-	
App3	0.668	0.588	-
App4	1.123	1.206	0.94

Table 3.1: Sample history records of a 4-application workload. The values are pair-wise sum of speedups of the applications in the corresponding rows and columns.

score returns 0. The same score is added for each application A_{ij} in a particular partition. Therefore, we divide the total score for each possible mapping by n, which is the number of applications in a partition. Once the application mapping matrix is found, a matrix multiplication is done with the topology matrix to find the application affinity matrix (Line 8).

The constraints (3.1)-(3.5) and the objective function in Algorithm 1, Line 7 formulate the application mapping problem as a set of linear equations. The algorithm requires to traverse a big number of application matrices. This number depends on the complexity of the shared resource topology and the size of the workload. Traversal is time consuming when workload size increases on complex configurations (e.g., multi-socket multiprocessor systems). The number of possible mappings increases exponentially as the number of applications increases [4]. To avoid a full traversal, we use *Binary Integer Programming* (BIP). Using the topology and application mapping matrices, BIP is used to solve the objective function *score*.

3.5 Example

We now go over an example to illustrate how an optimal mapping for a workload is calculated. Our goal is to maximize the weighted speedup of the workload running on a 4-core Harpertown machine. Suppose we try to find the optimal mapping for four applications, and the shared resource groups are defined based on cache topology. In this case, we have two resource groups, and the problem is to find application pairs to be mapped to each shared resource group.

Assume that we have a complete set of history records for our workload, represented as a matrix, as in Table 3.1. This table gives the *score* of each pair gathered from previous executions. The data collected is either from previous runs of this workload or different workloads that contained one or more subsets of the application pairs. For a workload of four applications, there are 3 different mappings: (App1, App2; App3, App4); (App1, App3; App2, App4); and (App1, App4; App2, App3). In this case, the mapping with the highest score is (App1, App2; App3, App4).

Note that our algorithm is not greedy. A greedy approach would pick the highest performing group first to map together (i.e., App2 and App4). Another approach to choose mappings could be avoiding groups (i.e., App2 and App3) that performs poorly. However, it would let us avoid worst case scenarios only. Also note that our algorithm does *not* require in general all possible application group combinations as in this case. It can make a prediction even with missing records. Consequently, the mappings will, in general, be sub-optimal. However, when there are enough records, the prediction is quite accurate, which gives better mappings and have higher performance.

Chapter 4

Experimentation

4.1 Experimental Setup and Methodology

To demonstrate benefits of cross-run statistics, we choose our goal as optimizing *the throughput of the entire multicore system.* For this purpose, we consider two throughput-oriented metrics: *weighted speedup* (WSU) and *million instructions executed per second* ($\sum$ MIpS). **WSU** is a fair system-wide metric and requires solo-run performance of each application. $\sum$ **MIpS** is a raw indication of total throughput of the workload and it does not require any preliminary information about applications (hence, it can be calculated on-the-fly). Table 4.1 gives the details of the systems used, as well as default values of experimental parameters.

We randomly chose applications from SPEC2006 [29] to form workloads. There are two reasons behind our choice of single-threaded applications to form our workloads. First, the operating system is oblivious of the applications properties that are run. For OS, it is merely a problem of scheduling threads (execution contexts) whenever necessary and possible. For operating system, an application is a thread with its own context whereas threads of a multi-threaded applications share the context. The major difference between a multi-threaded application from a single-threaded one would be constructive sharing effects due to shared context. However, this is not an issue for our method since workload partitioning is abstracted by a global performance value, namely the score. Any constructive sharing benefits of a particular workload partition would be captured the same way destructive interferences are captured. For our algorithm, there is only a score of a specific set

Processor	Intel Xeon E5472 (Harpertown)
Core Frequency	2.67 GHz
Number of Cores	4 per socket
Number of Sockets	2
Hyperthreading (SMT)	No
L1 Cache	32KB Private Data and Instruction caches
L2 Cache	2x6MB Dual-shared unified cache
Memory	64GB
Workload Sizes	6, 8
Processor	AMD Opteron 6174 (Magny-Cours)
Core Frequency	2.2 GHz
Number of Cores	12 per socket
Number of Sockets	1
Hyperthreading (SMT)	No
L1 Cache	32KB Private Data and Instruction caches
L2 Cache	12x512KB Private unified cache
L3 Cache	2x6MB Hexa-shared unified cache
Memory	64GB
Workload Sizes	12
Randomization Threshold	7
Adventure Threshold	0.1
Mapping Interval	5 seconds

Table 4.1: Major system configuration and experimental parameters.

of threads when they are sharing the same resources. Identification of individual threads or applications and keeping history records for each thread/application is simply an engineering problem and is not discussed in this work. Second, there were no particular multi-threaded benchmark application at the time of experimentation to stress the real machines. Popular multi-threaded application benchmarks such as PARSEC and NAS has very little memory footprint and were not sufficient to illustrate impact of contention on such systems. We have observed that, when the amount of contention is too little (or similarly too high), there is little to no benefit to deploy a mapping strategy since the performance difference among mappings is insignificant.

The workload compositions are given in Tables 4.2-4.4 We randomized workload selection to avoid bias toward certain workload types, as well as to make a fair and

Type	**Applications**
6H	*bzip2, GemsFDTD, h264ref, libquantum, mcf, namd*
6H	*bwaves, GemsFDTD, h264ref, milc, povray, sphinx3*
6H	*bwaves, bzip2, namd, omnetpp, soplex, sphinx3*
6H	*libquantum, mcf, milc, namd, omnetpp, povray*
6H	*bwaves, bzip2, GemsFDTD, h264ref, omnetpp, soplex*
6H	*bzip2, h264ref, libquantum, mcf, soplex, sphinx3*
6H	*bwaves, GemsFDTD, milc, namd, povray, sphinx3*
6H	*GemsFDTD, h264ref, namd, omnetpp, povray, sphinx3*
6C0	*libquantum, mcf, milc, omnetpp, povray, soplex*
6CR	*bwaves, bzip2, libquantum, mcf, milc, soplex*
6C1	*bwaves, libquantum, mcf, milc,* **sjeng**, *soplex*
6C2	*bwaves, libquantum, milc,* **sjeng**, *soplex,* **zeusmp**
6C3	**hmmer**, *libquantum, milc,* **sjeng**, *soplex,* **zeusmp**
6C4	**gromacs**, **hmmer**, *milc,* **sjeng**, *soplex,* **zeusmp**

Table 4.2: Application mixes for 6-application workloads. The non-familiar applications are bolded.

Type	**Applications**
8H	*bwaves, GemsFDTD, h264ref, hmmer, lbm, milc, omnetpp, zeusmp*
8H	*bwaves, calculix, GemsFDTD, h264ref, hmmer, milc, sjeng, sphinx3*
8H	*calculix, GemsFDTD, h264ref, mcf, milc, sjeng, sphinx3, zeusmp*
8C0	*bwaves, calculix, h264ref, lbm, mcf, omnetpp, sphinx3, zeusmp*
8CR	*GemsFDTD, h264ref, mcf, milc, omnetpp, sjeng, sphinx3, zeusmp*
8C1	*GemsFDTD, h264ref, mcf, milc, omnetpp,* **povray**, *sjeng, zeusmp*
8C2	*GemsFDTD,* **gromacs**, *h264ref, mcf, milc,* **povray**, *sjeng, zeusmp*
8C3	*GemsFDTD,* **gromacs**, *h264ref, mcf, milc,* **povray**, *sjeng,* **soplex**
8C4	*GemsFDTD, h264ref,* **leslie3D**, **libquantum**, *mcf, milc,* **povray**, **tonto**
8C5	*GemsFDTD,* **gromacs**, **leslie3D**, **libquantum**, *mcf, milc,* **povray**, **tonto**
8C6	*GemsFDTD,* **gromacs**, **leslie3D**, **libquantum**, *milc,* **povray**, **soplex**, **tonto**

Table 4.3: Application mixes for 8-application workloads. The non-familiar applications are bolded.

Type	Applications
12H	*bzip2, calculix, gobmk, hmmer, lbm, leslie3D, libquantum, milc, namd, omnetpp, sjeng, sphinx*
12H	*bzip2, gobmk, hmmer, lbm, leslie3D, libquantum, milc, namd, omnetpp, sjeng, soplex, tonto*
12H	*bwaves, bzip2, calculix, hmmer, lbm, libquantum, namd, omnetpp, soplex, sphinx, tonto, zeusmp*
12H	*bwaves, bzip2, gobmk, hmmer, lbm, leslie3D, milc, omnetpp, sjeng, soplex, sphinx, tonto*
12C0	*bwaves, bzip2, calculix, gobmk, lbm, leslie3D, libquantum, omnetpp, sjeng, sphinx, tonto, zeusmp*
12CR	*bwaves, bzip2, calculix, gobmk, hmmer, lbm, leslie3D, libquantum, milc, namd, omnetpp, sjeng*
12C1	**astar**, *bwaves, bzip2, calculix, gobmk, lbm, leslie3D, libquantum, milc, namd, omnetpp, sjeng*
12C2	**astar**, *bwaves, bzip2, calculix,* **GemsFDTD**, *gobmk, lbm, leslie3D, libquantum, milc, omnetpp, sjeng*
12C3	**astar**, *bwaves, calculix,* **GemsFDTD**, *gobmk,* **gromacs**, *lbm, leslie3D, libquantum, milc, omnetpp, sjeng*
12C4	**astar**, **GemsFDTD**, *gobmk,* **gromacs**, **h264**, *lbm, leslie3D, libquantum, milc, omnetpp, sjeng*
12C5	**astar**, **GemsFDTD**, *gobmk,* **gromacs**, **h264**, *lbm, leslie3D, libquantum,* **mcf**, *milc, omnetpp, sjeng*
12C6	**astar**, **GemsFDTD**, *gobmk,* **gromacs**, **h264**, *leslie3D, libquantum,* **mcf**, *milc, omnetpp,* **povray**, *sjeng*

Table 4.4: Application mixes for 12-application workloads. The non-familiar applications are bolded.

broad analysis.

We give each workload a unique ID, describing its size and type. We used workload sizes of 6 and 8 applications to run on Harpertown and 12 applications on Magny-Cours architectures. Workload size is prefixed to each workload ID. We also classify the workloads into two types: *history workloads* and *control workloads. History workloads* are used to generate history records but not for performance comparison. These workloads are formed from 20 randomly selected benchmarks. They are indicated using an *H. Control workloads* are run after a certain number of history records are collected that satisfy randomization threshold requirements.

They are indicated using a *C*. There are three types of control workloads:
• A workload chosen from the *history workloads* used as a control workload(e.g., *6C0*).
• The *root control workload* (e.g., *8CR*), which consists of applications that have been previously executed in one of the history workloads (i.e., *familiar* applications). It differs from all previous history workloads by at least one application.
• *Derived control workloads*, which are generated from the *root control workload* by randomly replacing familiar applications with those that have *not* been previously executed (i.e., *non-familiar* applications). Derived control workload IDs are suffixed by an integer (e.g., 12C**2**) which is the number of familiar applications replaced in the root control workload. Non-familiar applications and familiar applications to be replaced are selected randomly.

We collected architectural statistics with *perf* [30]. Mapping interval is selected to be 5 seconds. There are two reasons for such a coarse interval period. The first is the high variation of the statistics collected over small periods of time. This is mainly due to the overhead of performance monitoring in real machines. Performance monitoring interfaces are implemented in software. The interface provided in Linux is too slow to accurately capture different phases of an application, which is usually in degree of microseconds. The performance monitoring, as of the time this work has been performed, cannot measure such fine granular data without incurring significant overhead. As the interval time increased, we observed that the performance variations among different intervals of applications has decreased. Therefore, we chose relatively longer mapping intervals to amortize the overheads of the monitoring. We ran the workloads for 20 and 10 intervals on Harpertown and Magny-Cours, respectively. This translates into a total execution time of 100 seconds and 50 seconds for each mapping run. We stopped the experiments immediately for the workloads in which one of the applications has finished execution.

History data contains performance information of application pairs in Harpertown and 6-application groups in Magny-Cours. This is because we have chosen last-level cache as the shared resource in focus. However, our scheme is not tied to a particular shared hardware unit, hierarchy of resources or size of workload. In fact, the implementation can be extended to cover a variety of hierarchies of shared resources. By simple modifications to our application mapping matrix and

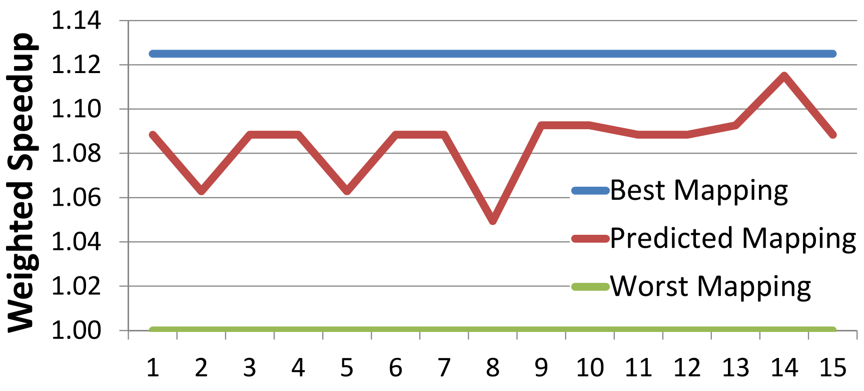


Figure 4.1: Performance of workload 6C0 over consecutive mapping intervals.

shared resource matrix models, our method can handle a vide range of hardware and workload configurations. Nevertheless, we keep our focus on a single crucial shared resource on a homogenous and symmetric CMP with a balanced workload for simplicity.

To generate the initial history data, we selected a random *history workload*, ran it for 5 intervals and repeated this process until the number of history records reached the randomization threshold. We ran 24 intervals for 6-application workloads, 30 intervals for 8-application workloads and 42 intervals for 12-application workloads to gather enough number of history records. The time (or number of mappings) to reach the desired number of history records depends on the total number of applications that will be used in all workloads. Therefore, if the number of applications that are run on a system is large, it would take a substantial amount of time to achieve the required threshold and get the benefits of history records. We discuss possible approaches to mitigate this problem in Chapter 5.

4.2 Experimental Results

Performance of the Algorithm:

The first set of results illustrate how performance relates to number of cross-run statistics collected for a single workload over the time. We present system performance observed using two workloads of 6- and 8-application workloads. Starting with the set of records collected from history workloads, we ran our algorithm for

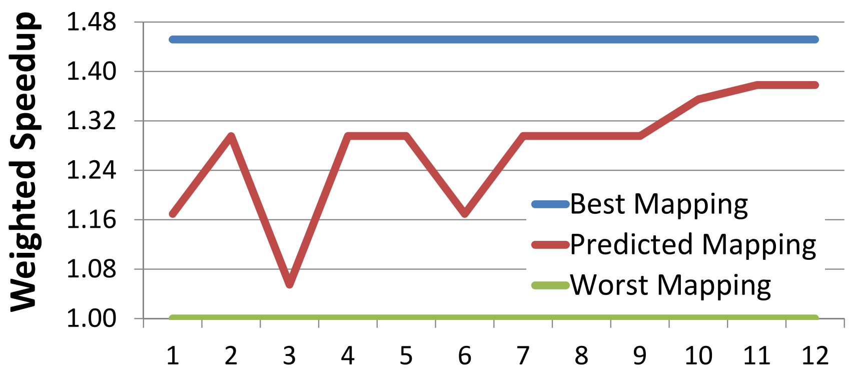


Figure 4.2: Performance of workload 8C0 over consecutive mapping intervals.

consecutive mapping intervals of each workload. For each interval, the algorithm tried to find a good mapping based on available data. It updated the history data at the end of each interval, before running the Algorithm 1 again. The results for each workload contain at least one randomization, that is, the mapping is found pseudo-randomly due to the adventure threshold. The corresponding plots for the workloads 6C0 and 8C0 are presented in Figures 4.1 and 4.2.

The workloads 6C0 and 8C0 have only familiar applications. Each instance on the horizontal axis of Figures 4.1 and 4.2 is the mapping selected by our algorithm based on previous mapping interval. As shown, the mappings found by the algorithm approaches to optimal, while making bad decisions along the way. This pattern is observed until either the performance score of the workload is accurate enough that the undesired mapping is not selected, or a random mapping is selected due to the adventure threshold.

We do not consider randomized mappings as good or bad, since they are random. However, randomization is important because it reveals better mappings and enables improved performance. The importance of randomization can be seen in 6C0 (Figure 4.1). The performance of 6C0 at the end of first mapping interval, in which all application are familiar, have low performance. The reason is the missing cross-run data for the highly-cooperative application pairs. Randomization helps to find such critical information to avoid getting stuck.

The algorithm usually finds a sub-optimal mapping over several intervals until a better mapping is revealed by randomization. The reason for this behavior is the performance difference among different mappings. Once the algorithm finds a good

Number of Familiar Applications	12	13	14	15	16	17	18
Fraction of Familiar Applications	60	46	43	33	31	24	22
Randomization Threshold	7	6	6	5	5	4	4

Table 4.5: Randomization thresholds (i.e., average number of records per application required) for various sizes of familiar applications with a fixed number of records.

mapping, it is less likely to find a better mapping with the given history records. Since the algorithm chooses the same mapping every time the workload is executed, it does not find data for new application pairs. Again, without randomization, the algorithm would be stuck at a fixed point.

Another interesting case for randomization is workload 8C0 in Figure 4.2. For this workload, randomization is triggered three times at intervals 3, 6 and 10. Mappings 3 and 6 are worse than the previous mapping. Yet, they have no effect on the algorithm's decision on its next invocation, because they do not add new records that reveal better co-operating application pairs. At instance 10, however, the randomization picks a better mapping by revealing a new pair. After finding a good mapping, the algorithm finds a better mapping (8% gain) than the previous one at instances 11 and 12. This example is another indication of the importance to collect as many history records as possible.

Sensitivity to Randomization Threshold:

The next results are from experiments with the control workloads illustrating benefits of our algorithm for two metrics of performance. Similar to the previous experiment, these workloads are executed with a fixed set of history records which are updated on-line at the end of each mapping interval.

For control workloads with non-familiar applications, our algorithm would select a mapping randomly because the number of records per application is less than the defined randomization threshold (i.e., 60% of the number of the familiar applications). Nevertheless, we choose the mapping that provides the highest score to see the impact of the number of records on the performance of the mapping found by the algorithm. Control workloads with non-familiar applications demonstrate sensitivity of the algorithm to the randomization threshold. With a fixed set of history data, the average number of records per application decreases

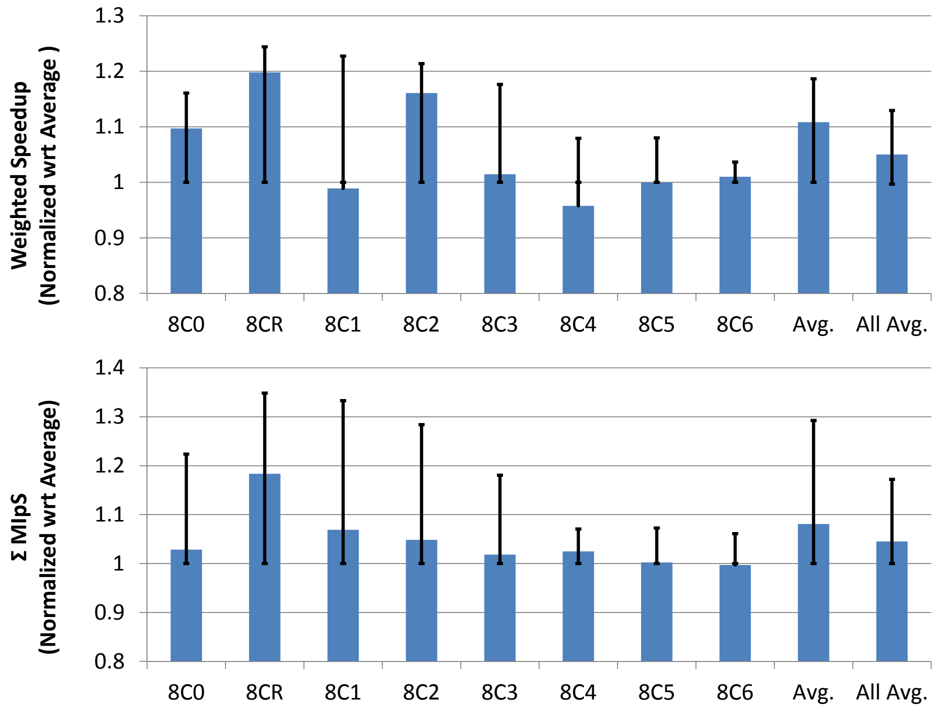


Figure 4.3: Performance for 8-application workloads. Error bars show maximum and average case performance at the last mapping interval. *Avg.* is average of workloads with at most 2 non-familiar application. *All Avg.* is average of all workloads.

as the number of non-familiar applications increases in a workload. Therefore, running a control workload with non-familiar applications is similar to running the algorithm with different randomization thresholds when all applications are familiar. The randomization thresholds for different number of applications are given in Table 4.5.

The performance for 8-application workloads for both performance metrics are given in Figure 4.3. The bars indicate performance gain at the very last interval executed. They represent the normalized weighted speedup over the average case. The error bars indicate the performance difference between the best and the average case mappings. The average case is calculated by averaging performance of all mappings for that interval. Average performance of all possible mappings is the expected performance of the Linux OS scheduler. This is because the Linux OS scheduler aims to maximize core time utilization as well as data locality. Since we assume that there are as many applications running as the number of cores,

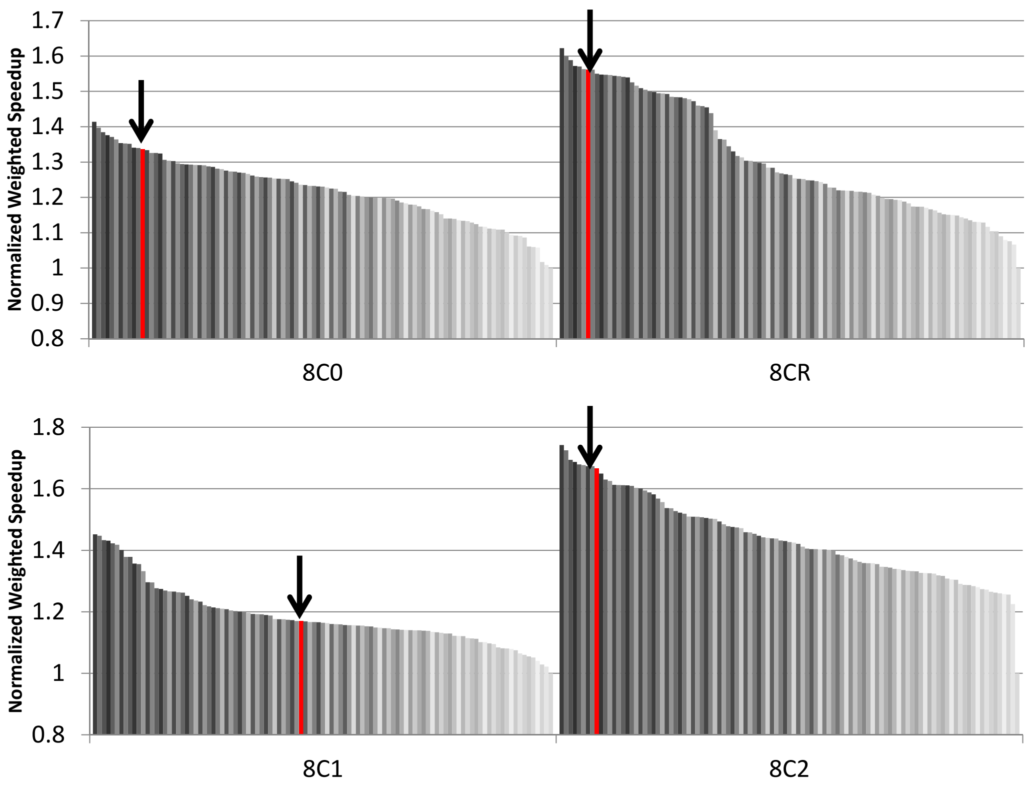


Figure 4.4: WSU of each possible mapping for the 8-application workloads at the last mapping interval.

the scheduler aims to utilize all cores. Moreover, to preserve locality, OS scheduler mappings are not changed throughout workload execution. Since OS scheduler mappings are random in nature, it is safe to assume that OS scheduler will be close to the average of all possible mappings. Therefore, we chose average performance of all mappings at a each interval for comparison.

The results show that, in most cases, the mapping found by our algorithm is significantly better than a random mapping scheme for both performance metrics. The control workloads with familiar applications outperform the average case by up to 20%. When number of non-familiar applications is low, the algorithm finds mappings with 11% better WSU on average. This illustrates that the Overall, it provides 6% improvement over the default Linux scheduler for all workloads when WSU is used. Moreover, the algorithm finds a mapping for workloads consisting of familiar applications that is within 5% of the optimal mapping, among all 105 possible mappings.

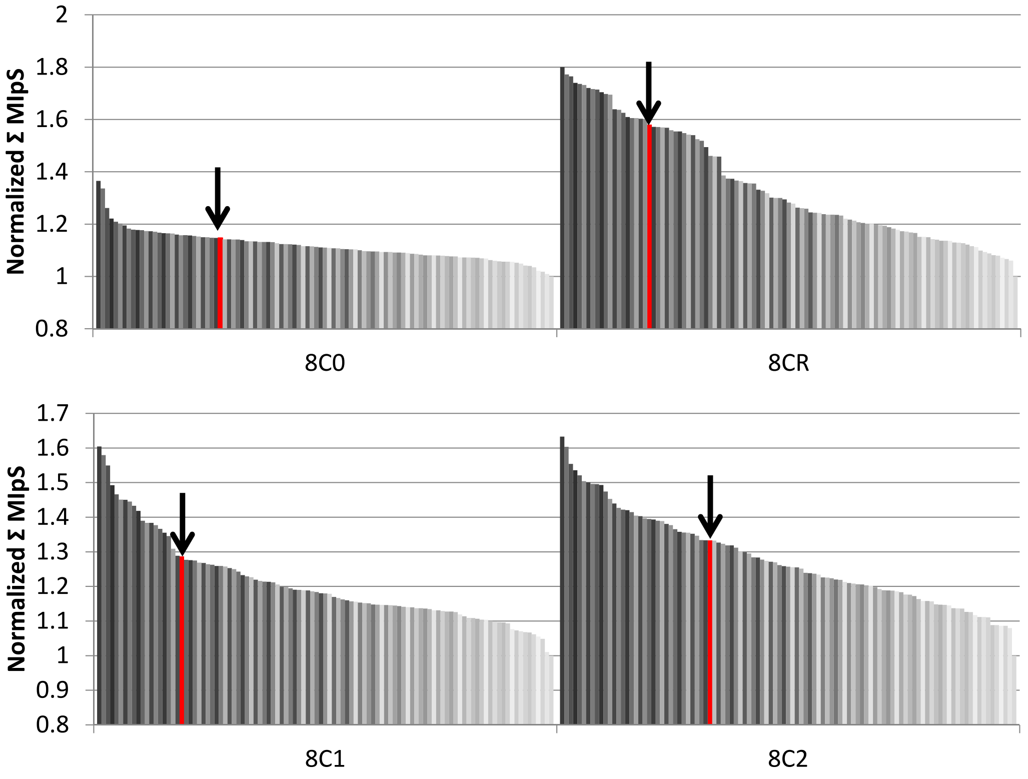


Figure 4.5: $\sum$MIpS of each possible static mapping for the 8-application workloads at the last mapping interval.

Figures 4.4-4.7 show relative performance of 8 and 12 application mappings for the same experiment for WSU and $\sum$ MIpS. For a given workload in the horizontal axis, each bar corresponds to a static mapping performance at the last interval executed. The mappings are sorted based on weighted speedup and normalized against the worst-case mapping. The mapping selected by our algorithm is indicated by a red bar and an arrow.

As seen in Figures 4.6-4.7, the performance gap between best and worst cases of 12-application workloads is small (7% at most). In fact, the best mapping is only 3% better than the average case at most. This implies that, even with a mapping scheme that chooses best mapping for a given interval, there are cases (i.e., workloads) that will yield minimal benefits for using that scheme. Considering the monitoring and re-mapping overheads, it might even hurt the performance significantly. The phenomena seen in these examples are partly because of the extensive contention over memory bandwidth. In the Magny-Cours architecture, each L3 cache block and memory controller serves 6 applications. The amount of contention creates bottleneck at crucial resources (i.e., cache space, on-chip bus

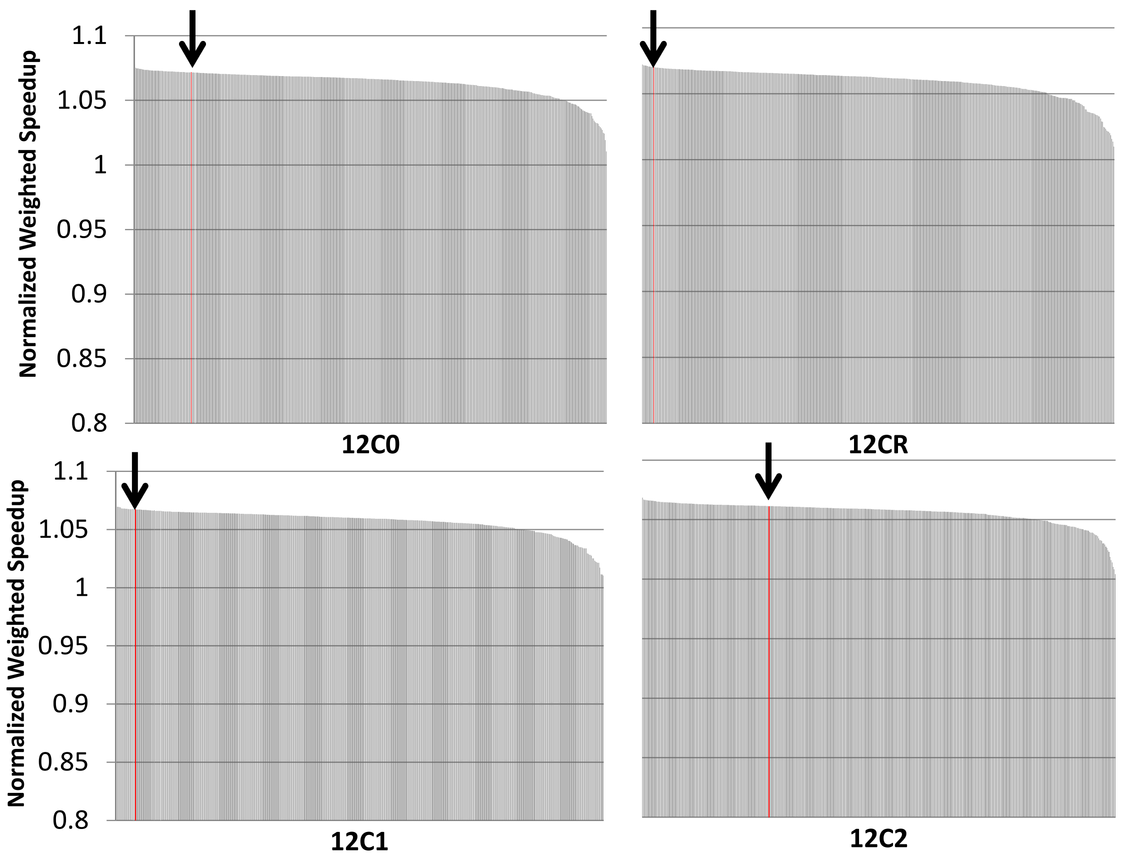


Figure 4.6: WSU of each possible static mapping for the 12-application workloads at the last mapping interval.

bandwidth and off-chip memory bandwidth) that have major impact on throughput of the overall system. As seen in the figures, there are only few mappings which perform relatively worse (due to making the contention problem worse by grouping adversely affected applications together to use same set of resources). We believe that, in such cases, it is sufficient enough to identify the problematic cases and avoid them. Our approach is capable of avoiding worst case scenarios as it is. We discuss how we can adapt our scheme to such cases to potentially avoid overheads in Chapter 5.

4.3 Observations and Discussions

Overhead: The overhead associated with the scheme introduced comes in 2 forms:

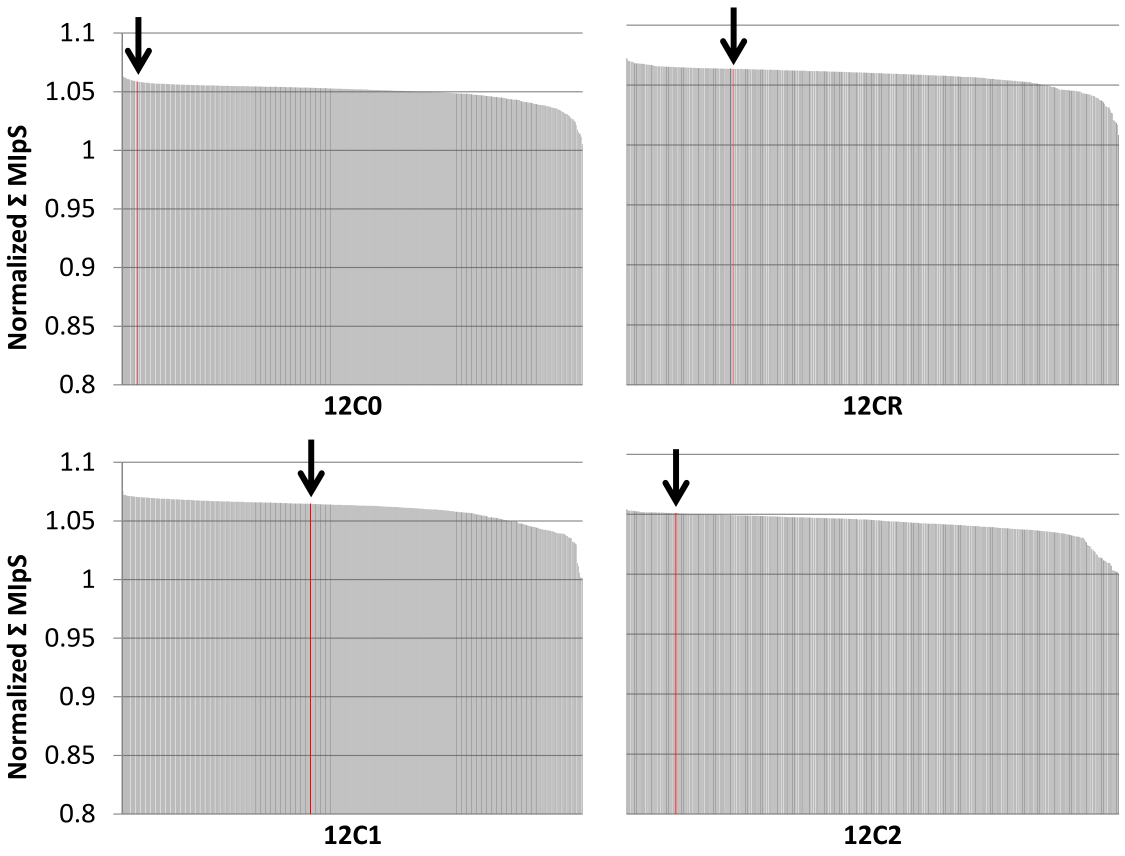


Figure 4.7: $\sum$MIpS of each possible static mapping for the 12-application workloads at the last mapping interval.

Algorithm's execution time and storage of history data.

As stated before, our algorithm is an off-line scheme. Therefore, our solution does not incur any runtime for the applications. The time spent to find the optimal mapping is less than 2% of the total execution time of the workload.

Each entry in history data consists of a group of application IDs and the score (of the group) that needs to be optimized. We keep the history data on secondary storage. The amount of data we keep per application group is much less than 1KB. Therefore, it is unlikely that keeping history data will be a problem for real systems.

Sensitivity to Application Input Variations:

Most applications' performance depends on the data they work on. Since history data is closely related to application behavior, it is important to evaluate ef-

	astar	*bzip2*	*hmmer*	*soplex*
bwaves	2%	23%	2%	9%
calculix	1%	21%	1%	3%
GemsFDTD	3%	26%	2%	8%
gobmk	2%	22%	0%	0%
gromacs	0%	21%	0%	0%
h264ref	1%	23%	0%	1%
lbm	0%	23%	2%	1%
leslie3D	3%	27%	2%	9%
libquantum	5%	27%	3%	8%
mcf	3%	24%	0%	9%
milc	5%	24%	2%	7%
namd	1%	21%	1%	0%
omnetpp	2%	24%	1%	6%
povray	0%	22%	1%	0%
sjeng	0%	21%	1%	1%
sphinx3	3%	25%	3%	9%
tonto	1%	21%	1%	0%
zeusmp	0%	21%	1%	4%

Table 4.6: Difference in scores of application pairs with different input set size. Applications on the top row have at least two input sets provided. They are co-run with applications on the leftmost column for each input set they have. Results presented are percent difference of pairwise performance with different input sets.

fects of varying inputs of applications on the history data. For this, we performed a preliminary study. We run the applications used in our experiments in pairs with other applications that have different input sets provided in the SPEC2006 on Intel Harpertown machine. The applications with different input sets (i.e., *astar, bzip2, hmmer* and *soplex*) are co-run with other applications for each set of inputs. We compare the pairwise score of varying input sets. The difference in the scores is presented in Table 4.6. These results suggest that the score metric (i.e., weighted speedup value) has less impact then one would expect for some applications. The insignificant difference in the scores of the different input sets of *astar, hmmer* and *soplex* can be explained by the coupled cache performance of the applications.

In our experiments, the main factor that shapes performance of co-running applications is the behavior of the shared last level cache usage. The weighted speedup value is the summation of each application's speedup. In other words, our algorithm considers groups of applications, and not the performance of individual applications. The individual performance of applications with different input sets vary significantly. However, many application groups together has an upper bound in throughput value due to the shared resource bottleneck. Therefore, the individual performance fluctuates with different input sets, but total performance of the group remains the same. An exceptional case is *bzip2*, which exhibits irregular cache access patterns depending on input. Its performance is not affected much by its co-runner with different input sets. Yet, it affects the performance of the co-runners if they are cache-intensive. The differences in pairwise performance for *bzip2* in Table 4.6 are mainly due to the varying performance of the coupled application.

Chapter 5

Conclusion and Further Discussions

In this work, we try to tackle the problem of how to map applications to hardware resources in a multicore architecture. Destructive interferences can lead to performance degradation when two or more applications contend for various shared hardware units. We start with providing a model to translate application mapping in multicores into an Integer Programming problem that covers various system and shared hardware configurations. Then, we introduce the concept of cross-run statistics along with its possible uses. Later, we propose a technique that exploits cross-run statistics to decide how to map applications when they are executed again in similar workloads. The scheme successfully determines highly co-operative application groups and maps them to the hardware resources in a way that mitigates contention. The experiments demonstrate that history data can be used to find near-optimal mappings, avoiding potential performance losses. Performance benefits are 11% and 8% on average compared to average case mapping for WSU and $\sum$ MIpS, respectively.

There are potential issues with the solution provided that need to be addressed in the future. First, we present results for a specific implementation that targets the last-level cache only for illustrative purposes. However, a study is required to cover a variety of shared resources at the same time. As it is, the proposed algorithm can be modified to handle various system and shared resource configurations at the same time with minimal work. When addressing several resource types, we need to find as many Application Mapping Matrices as the number of resources we need. Starting from the highest level of sharing (i.e., resources that are shared by the most number of application), each solution will yield a new set of workload partition, which need to be further partitioned as we go deeper in shared-resource hierarchy.

Second, our algorithm tracks individual applications. As discussed before, our algorithm gives the best results when there is sufficient amount of history records for each application. In a personal computer system, we could expect to have a reasonable number of applications that are frequently used (i.e., browser, anti-virus software, text editor, etc.) which will allow us to efficiently implement our history records. However, it is harder to gather the data when the number of frequently used applications is too many: Imagine a cluster system shared by a whole university! As the number of applications used in the system increase, the algorithm needs more randomized mappings to satisfy defined thresholds. This implies that the algorithm requires to randomly try many possible mappings, which would not happen within a reasonable time.

Lastly, the applications we use are from a benchmark suite where inputs and parameters are preconfigured and fixed. However, application characteristics vary drastically based on the inputs and parameters used for certain application types. Therefore, a history record created for one application with a given input set might not be the best indicator of performance when the input set is different.

A possible solution for the last two problems is to save records based on application/thread classes rather than individual applications using techniques similar to previous studies [21, 26]. Once applications are classified based on distinctive features, not only the amount of history data will be reduced (due to less number of classes than the number of individual applications), but also application characteristics depending on input sets will be abstracted. Application classification is a hard problem by itself and needs to be investigated further to reflect interactions among various shared resources. Since we have observed significant benefits of using history data, we believe it is a viable direction to make our scheme even more generic.

Other than the issues discussed above, we would like to elaborate on possible uses of our scheme with non-existing (yet implementable) techniques proposed in literature. Our solution is very attractive since it can be implemented with existing hardware and software capabilities of multicore systems. As the transistor sizes continue to scale, we expect to see more cores, deeper memory hierarchies and possibly hardware support for features that was not possible or feasible to implement before. The chip industry is already putting efforts in implementing better chips to address hard-to-tackle problems of multicore designs such as on-chip communication costs (networks on chip), off-chip communication costs (on-chip memory controllers, on-chip network modules), CPU-memory gap (deeper/bigger cache hierarchies). Moreover, architectures pack previously off-chip units (i.e., graphics units) into the very same chip to provide application-specific solutions (systems on chips) or to help mitigate the heterogeneity of vast variety of applications. But the question we would like to pose is: Will that be sufficient or the best way to handle heterogeneity of program characteristics? As long as the chip design pertains its fixed and symmetric nature, we have to find the "right" hardware for each application (or workloads) among the vast design space. This poses a challenge for both chip manufacturers and end-users, which increases the effort (hence the cost) of both hardware and software products. We are aware that it is very unlikely to come up with a generic design to serve all purposes, but we advocate the benefits of flexible designs (both hardware and run-time) that can adopt to dynamic nature of applications.

To illustrate the benefits of flexible design, we investigate our 12-application workload cases. As we mentioned previously, on-chip resources (last level caches and memory controllers) are choked down by 6 applications contending on them, hurting the performance of the overall system. In this fixed cache hierarchy design, it is not possible to prevent destructive interference from memory intensive applications while maintaining load balance and utilizing other available resources. For instance, we consider *lbm* which streams data from memory, hence highly memory intensive. This application would benefit from higher memory bandwidth, but not from bigger cache space. Basically, in its streaming phase, *lbm* kicks out cache data which could be potentially reused by other applications. To make the case worse, it also delays non-memory intensive applications' memory requests. The best way to deal with such applications is to isolate their cache and memory bandwidth usage from others to minimize the damage. Since performance of *lbm* is merely affected by the amount of cache provided, a good strategy would be to assign a small portion of cache space for this application only. This would detain the destructive cache interference resulting from streaming. To our knowledge, such isolation is not possible with the current hardware and systems software available on the market.

An interesting study has been proposed by Srikantaiah et al. [31] that would be useful for the problem addressed above. In their study, Srikantaiah et al. propose a reconfigurable cache hierarchy, namely MorphCache, to mitigate problems of "one-cache-topology-fits-all" philosophy. They provide a cache design which is capable of partitioning the cache on multiple levels and "morph" into asymmetric topologies to better address the heterogeneity of workloads. The topologies that can be derived are not arbitrary due to consistency and coherency concerns. Yet, MorphCache seems to be the most comprehensive solution at the time as a flexible CMP architecture design. However, MorphCache has its own limitations. The cache merging/splitting can only be performed among adjacent cache blocks and it is limited to merging/splitting a number of cache blocks that are a power of 2.

Considering the limitations of MorphCache, it is evident that it cannot completely mitigate destructive cache sharing effects. Nevertheless, it provides a better resource design infrastructure for our run-time system. In fact, our methodology provides a mutual run-time system to (possibly) improve the benefits of using MorphCache. Note that, with MorphCache, our system will be able to traverse a wider range of solutions for a given workload over time. MorphCache allows us to reduce the cache pollution affects by creating isolated cache space. While doing so, it can allocate smaller cache partitions to memory intensive applications while granting more cache space to cache-sensitive applications (compared to fixed 6-core shared last level cache design of AMD multicore). Furthermore, the cache blocks distributed to applications could be changed on the fly depending on the workload compisition, which is the main advantage of MorphCache.

MorphCache can benefit from a run-time system that is aware of the underlying architecture. By design, MorphCache is oblivious of the run time system. However, its performance is tightly coupled with the decisions of application mapping. As an example, assume a 4-core MorphCache architecture with 1 level of reconfigurable cache. Also assume that there are 3 applications (A, B and C) running on the system. Among these applications, A is a computation intensive application, B is a cache intensive application and C is memory intensive one. Ideally, you would like to map A and B to adjacent cores on our system to increase cache space while isolating C from others. However, if A and B are not mapped to adjacent cores, MorphCache has no means of modifying the mapping. It will instead try and split cache space into smaller chunks, possibly wasting cache space dedicated to A which could instead be used by B. Therefore, MorphCache is potentially prone to performance loss without a proper application mapping scheme.

With minimal modifications, our scheme can be adapted to MorphCache-like architectures easily. First modification is to handle various possible cache configurations. To achieve that, we can populate multiple topology matrices and solve the corresponding objective function for each possible system configuration. The problems are independent of each other, hence can be solved in parallel - which could hide potential overheads. The best solution among the possible solutions would be selected as the mapping and the choice of cache configuration. This will imply that we have to keep a separate history record database for each possible configuration. With more number of cores or deeper resource hierarchies, the number of possible configurations is expected to grow exponentially. Therefore, we need to find simple heuristics to reduce number of possible configurations. Together with a reconfigurable resource infrastructure such as MorphCache, our model can provide additional benefits for a wider range of workload types more efficiently.

Bibliography

[1] I. Corp. Harpertown Architecture (Intel Xeon 5400 Series). [Online]. Available: http://ark.intel.com/products/series/33905

[2] A. Corp. Magny-Cours Architecture (AMD Opteron 6100 Series). [Online]. Available: http://www.amd.com/US/PRODUCTS/SERVER/PROCESSORS/6000-SERIES-PLATFORM/Pages/6000-series-platform.aspx

[3] S. Chen *et al.*, "Scheduling threads for constructive cache sharing on CMPs," in *Proceedings of SPAA '07*, 2007, pp. 105–115.

[4] Y. Jiang *et al.*, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proceedings of PACT '08*, 2008, pp. 220–229.

[5] S. Siddha *et al.*, "Process Scheduling Challenges in the Era of Multi-core Processors." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[6] S. Cho *et al.*, "Managing distributed, shared L2 caches through OS-level page allocation," in *Proceedings of MICRO 39*, 2006, pp. 455–468.

[7] J. Lin *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proceedings of HPCA '08*, 2008, pp. 367–378.

[8] S. Srikantaiah, M. Kandemir, and M. J. Irwin, "Adaptive set pinning: managing shared caches in chip multiprocessors," in *Proceedings of ASPLOS '08*, 2008, pp. 135–144.

[9] S. Srikantaiah *et al.*, "A case for integrated processor-cache partitioning in chip multiprocessors," in *Proceedings of SC '09*, 2009, pp. 6:1–6:12.

[10] X. Zhang *et al.*, "Towards practical page coloring-based multicore cache management," in *Proceedings of EuroSys '09*, 2009, pp. 89–102.

[11] Y. Zhang *et al.*, "A simulation-based study of scheduling mechanisms for a dynamic cluster environment," in *Proceedings of ICS '00*, 2000, pp. 100–109.

[12] M. Bhadauria *et al.*, "An approach to resource-aware co-scheduling for CMPs," in *Proceedings of ICS '10*, 2010, pp. 189–199.

[13] A. Fedorova, M. Seltzer, and M. D. Smith, "Cache-fair thread scheduling for multicore processors." *Technical Report, Harvard University*, 2006.

[14] A. Fedorova *et al.*, "Performance of multithreaded chip multiprocessors and implications for operating system design." in *Proceedings of ATEC '05*, 2005.

[15] A. Fedorova, "Operating system scheduling for chip multithreaded processors." *PhD Thesis, Harvard University*, 2006.

[16] R. Knauerhase *et al.*, "Using OS observations to improve performance in multicore systems," *IEEE Micro*, vol. 28, pp. 54–66, May 2008.

[17] D. Tam *et al.*, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proceedings of EuroSys '07*, 2007, pp. 47–58.

[18] V. Kazempour *et al.*, "Performance implications of cache affinity on multicore processors," in *Proceedings of Euro-Par '08*, 2008, pp. 151–161.

[19] A. Snavely *et al.*, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of ASPLOS '00*, 2000, pp. 234–244.

[20] D. Chandra *et al.*, "Predicting inter-thread cache contention on a chip multiprocessor architecture," in *Proceedings of HPCA '05*, 2005, pp. 340–351.

[21] S. Zhuravlev *et al.*, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of ASPLOS '10*, 2010, pp. 129–142.

[22] Y. Z. et al., "A simulation-based study of scheduling mechanisms for a dynamic cluster environment," in *Proceedings of ICS '00*, 2000, pp. 100–109.

[23] D. Tam *et al.*, "RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations," in *Proceeding of ASPLOS '09*, 2009, pp. 121–132.

[24] R. McGregor *et al.*, "Scheduling algorithms for effective thread pairing on hybrid multiprocessors," in *Proceedings of IPDPS '05*, vol. 1, 2005.

[25] A. Fedorova *et al.*, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *Proceedings of PACT '07*, 2007, pp. 25–38.

[26] Y. Xie *et al.*, “Dynamic classification of program memory behaviors in CMPs.” *2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects. CMP-MSI ’08*, 2008.

[27] E. Koukis *et al.*, “Memory bandwidth aware scheduling for SMP cluster nodes,” in *Proceedings of the 13th EUROMICRO ’05*, 2005, pp. 187–196.

[28] Y. Kim *et al.*, “ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers.” in *Proceedings of HPCA ’10*, 2010, pp. 1–12.

[29] S. Performance Evaluation Corporation (SPEC). SPEC CPU2006 benchmark suit. [Online]. Available: http://www.spec.org/cpu2006/

[30] P. C. for Linux. Performance counters for Linux Wiki. [Online]. Available: https://perf.wiki.kernel .org

[31] S. Srikantaiah *et al.*, “MorphCache: A reconfigurable adaptive multi-level cache hierarchy,” in *Proceedings of HPCA ’11*, 2011.