ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Dynamic Resource (Cache and Power) Management in CMPs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Χριστίνας Σ. Δελημήτρου

Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

Επιβλέπων: Νεκτάριος Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2009.

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ
ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

Dynamic Resource (Cache and Power) Management in CMPs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χριστίνα Σ. Δελημήτρου

Επιβλέπων: Νεκτάριος Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2009

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ
ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

Dynamic Resource (Cache and Power) Management in CMPs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χριστίνα Σ. Δελημήτρου

Επιβλέπων: Νεκτάριος Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Ιουλίου 2009.

……	……	……
Ν. Κοζύρης	Δ. Σουντρής	Γ. Οικονομάκος
Αν. Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2009

……

Χριστίνα Σ. Δεληµήτρου

Διπλωµατούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Καθώς τα υπολογιστικά συστήματα γίνονται πιο πολύπλοκα και οι εκτελούμενες εφαρμογές πιο απαιτητικές σε υπολογιστικούς πόρους εμφανίζεται η πρόκληση της αξιοποίησης των διαθέσιμων υπολογιστικών πόρων με όσο το δυνατόν αποδοτικότερο τρόπο ώστε να προσαρμόζεται στις εκάστοτε ανάγκες των προγραμμάτων πολυπύρηνων και πολυεπεξεργαστικών συστημάτων. Η κύρια συμβολή της παρούσας εργασίας αφορά στην παρουσίαση και σύγκριση διαφορετικών πολιτικών διαμοιρασμού της κοινής μνήμης (cache) μεταξύ ταυτόχρονα εκτελούμενων νημάτων (threads) διατηρώντας μια μέριμνα ως προς την κατανάλωση ισχύος που αυτές οι πολιτικές εισάγουν στο σύστημα. Συγκεκριμένα, παρουσιάζουμε τρεις πολιτικές που βασίζονται είτε σε διαμοιρασμό της κρυφής μνήμης σε επίπεδο συσχετιστικότητας είτε σε διαμοιρασμό της σε σχέση με τον τρόπο που τα δεδομένα εισάγονται σε αυτήν. Μια εκτενής παρουσίαση αυτών των μεθόδων, μια μεταξύ τους αξιολόγηση καθώς και η πρόταση μιας νέας μεθόδου διαχείρισης της κοινής μνήμης αποτελούν τους κύριους άξονες της παρούσας εργασίας. Για την πραγματοποίηση των πειραμάτων χρησιμοποιήθηκε ένας προσομοιωτής υψηλού επιπέδου, για την ελαχιστοποίηση του απαιτούμενου χρόνου, ενώ οι εφαρμογές που χρησιμοποιήθηκαν είναι ερωτήσεις (queries) σε βάσεις δεδομένων και εφαρμογές εξυπηρετητών της σουίτας SPEC*Web*99.

Abstract

The main purpose of this thesis is to address the issue of resource management in multicore and multiprocessor systems. As chips become more complex and applications more computing intensive it comes to the computer architects to find a way to manage the different types of resources between the competing threads concurrently executing in a CMP system. The main contribution of this work is to define ways to manage the shared last level cache (LLC) between competing applications while maintaining a concern on the power consumption of the system. As far as cache memory is concerned different types of cache partitioning policies are explored and a review of their results is presented as well as a comparison between them. More specifically, we present three cache management schemes, relying either on partitioning or insertion algorithms to distribute the ways of an associative cache to competing threads. In order for the results of these schemes to be tested a high-level simulator has been used. Simflex, a module developed in extension of Simics provides a high-level abstraction of simulation that minimizes the required simulation time. The results are evaluated based on benchmarks from the SPEC*Web*99 benchmark suite.

Contents

Acknowledgements

As I am concluding my studies at NTUA there are some people that I would like to thank for their support, their guidance and for making these past five years some of the best in my life so far.

First of all, I would like to thank my advisor Prof. Koziris for his support and advice throughout the last year of my studies. For always finding time to hear my concerns, give me invaluable consults exaclty when I needed them and for supporting my decisions for both my current and future studies. I am indebted.

I also wanted to thank Kostis Nikas for co-advising my thesis and for always being eager to hear my ideas, my concerns and for his patience throughout our discussions. I hope I haven't been too much of a trouble…

I want to thank Prof. Soundris and Prof. Economakos for serving as members of the committee for the examination of this work.

To my friends and colleagues at NTUA a big thank you, for always being there to support me, stand by me and for binding your memories with mine. You are the best people to bound off ideas anyone can hope for. I will miss our long brainstorming sessions.

Finally, I would not have been here had it not been for the love, caring and support of my family and specifically my parents. Although they will probably not understand much from this work, I know they are proud of me.

List of Figures

Chapter 1

Introduction

"Every new beginning comes from some other beginning's end."

Seneca

Computer systems today trying to meet up with the increasing demands of computing intensive applications focus on advancing their performance. In order for that to be possible the approach deviates from uncontrolled sharing of computing resources and in its place a more efficient management of them is necessary. In that direction, a large number of research papers has developed ways over the years to address this issue effectively. The effort to keep producing systems with increased bulk performance can be to no avail since other restrictions rise either related to the actual hardware resources or to the power consumption, with the latter being one of the greatest challenges computer architects face today. Moreover, it becomes clear that the management of them can no longer be static. Realizing that, we get to consider monitoring aspects of the system, so that this resource management will be contigent upon the applications' behavior and needs.

In this work, we attempt to present a dynamic approach to handle and share the computing resources between competing programs executing in a Chip Multiprocessor System (CMP) system, one that will more than just affect the performance of the final system, succeed in the cutting-edge task of exploiting the available resources to the point where both each thread and the total system present an optimized performance. Apart from an increase in performance, however, resource management derives from other requirements targeting scheduling issues, security and concurrency in CMPs.

The main contribution of this work lays in the extensive and detailed study of

shared cache management schemes with a concern on the power consumption they issue. We offer a comparative study on the effectiveness of these techniques both qualitatively and quantitatively. Those schemes, have to this point been evaluated for small systems (up to 8 cores). We extend this study by performing experiments for larger systems of 16 cores.

More analytically all these issues will be presented in the following chapters of this thesis.

The way this work is organized is as follows : Chapter 1 consists of an Introduction to the area of resource management in CMPs and of the various challenges presented as part of it. Chapter 2 presents an overview of the problem and related work on cache partitioning. In Chapter 3 we find a description of the used tools and workloads, while Chapters 4-7 present different cache management policies relying either on cache partitioning policies or adaptive insertion schemes. If the reader is looking for a short overview of the proposed schemes he can skip to Chapter 8. Chapter 8 is a recap of previous chapters and covers an evaluation of the different policies presented earlier. Chapter 9 addresses the effect cache management schemes may have on the energy efficiency of the system, while Chapter 10 consists of various future research interests on the same or closely related topics. Finally, Chapter 11 refers to concluding remarks. The results are evaluated based on Web Server and Databases multithreaded benchmarks from the SPEC*Web*99 benchmark suite.

Chapter 2

Overview of the Problem

"The beginning is the most important part of the work."
Plato

2.1 Overview of the Cache Management Issue

The targeted work in this thesis is the efficient management of shared resources among competing applications in a multithreaded and/or multiprocessor system. As a first resource, shared caches are examined. This subject has come in the center of research on memory systems over the past years. As systems become more complicated and applications more resource consuming the available cache space needs to be effectively partitioned, taking into consideration the special features of applications, so that an optimal utilization of it comes at a minimum cost. In this section we present related work on this subject and pose the goals for the work presented in the following sections.

First of all, we must make a quick note related to the main approaches followed when facing with the subject of resource management (exceeding cache management) in a CMP system [HRI06]. These approaches vary on the goals that they set targeting the management of shared resources.

The main ways to address the cache partitioning problem as in every other resource management area are the:

- **Capitalist**

According to this attitude, applications are appointed computing resources in a demand-based policy which gives to the application with the highest demand the most resources in the system. This however does not take into account fairness requirements between competing threads and can therefore not consist a robust and tenable solution to the resource management issue. It is mainly adopted in systems running highly prioritized applications which should have increased privileges in the various resources of the system.

- **Communist**

Another approach to this issue, proposes the exact opposite solution by fair sharing of all resources among competing threads. Ideal though it may seem, this approach has some negative effects on the system's performance, since giving equal resources to each application implies equal needs. However, that is not always the case. This results in more computing intensive applications being resource deprived while others sparing resources they don't need. It becomes clear that either one of these approaches is not efficient enough since it doesn't exploit any information of the special features of the executing applications.

- **Elitist**

Since the two extreme approaches of the previous paragraphs have been proven inefficient a more median solution has been proposed. Granting priorities to some of the performing applications, the elitist approach in resource management, promotes their execution by giving them more resources than in the rest of the competing applications. This appears to be a more efficient solution, taking into consideration both fairness and priorities requirements and as such is applicable in a

number of cases, though it can still lead to a number of problems (such as thread starvation or unnecessary priorities). In all the previous proposals we have seen that the main goal is the performance improvement of specific applications rather than that of the whole system. There is however an alternate approach to deal with this issue.

- **Utilitarian**

The utilitarian approach rather than focusing on specific applications' behavior as a result of the resource management targets the improvement in the performance of the overall system. To do so, it may boost or cut back the execution of some threads, if doing so improves the overall IPC.

None of these approaches, except for the last one relies on the threads' execution patterns to define the partitioning of resources.

All these approaches rely for their decisions primarily on high-level priority vs. fairness issues without taking into consideration - except in some degree for the utilitarian approach - the patterns in threads' execution. All of them, can also be addressed as either static or dynamic allocation mechanisms, with dynamic – as will be proven by the results – being a clear choice among the two.

In the current work we focus mainly in the two last ways either by promoting the execution of high priority applications – where priority from now on will be considered as the benefit a thread can have from the cache - or by focusing in the improvement of the overall IPC (weighted IPC) of the computer system. The decision between these perspectives is entirely at the jurisdiction of the computer architect and is contigent upon the needs and prerequisites of the computer system.

Each of these approaches has been addressed by a series of research work the main points of which will be presented here. We must note however that most of the research papers so far, have concentrated on improving the performance of each of

the executing applications (IPCs) and few of them have focused in the increase in the weighted – overall IPC of the system. In all cases, these schemes maintain a monitoring process to evaluate and exploit the threads' execution pattern to appoint efficiently the available computing resources to them and a mechanism that will actually perform the sharing of the cache to achieve one of the above targets. Although the last two approaches (elitist and utilitarian) are more efficient and adjusted to the applications' needs, there are cases where each one of all four approaches can be proven optimal. Therefore, we should not be hasty when deciding among them on which one to adopt.

2.2 What are the Goals and the Means?

As computer systems become more complex and multithreaded applications more computing intensive efforts focus on effectively managing the shared last level cache (LLC) to avoid underutilization or thrashing effects in its use. Since low miss rates on the LLC leads to reduced latencies from main memory accesses it becomes clear why the issue of successfully managing it is considered as very important. Furthermore, realizing that these fewer latencies will result in lower power consumption demonstrates that the benefits from this management are not one dimensional.

2.3 Related Work

In this direction many research ideas have been proposed. The two that prevail in the cache management area are *cache partitioning policies* and *adaptive insertion policies*. Each one of them, comes with certain advantages and of course is

accompanied by a number of disadvantages, so that the final decision on their adoption is completely at the computer architect's jurisdiction. Both these schemes are presented in detail in the following chapters and are therefore not repeated here.

However, those where not the first approaches on the subject of managing the shared cache. The work by Sohi and Loh preceeded offering approaches to partition the cache on a way-granularity.

Furthermore, it useful to note here that way-partitioning is not the only way to see the issue of managing a shared cache. Other approaches consist of capacity based and priority based partitioning, while the actual partitioning of the cache, in the form of a NUCA structure is also a viable solution attracting attention over the years.

Each of the partitions is appointed to a core and a coherence protocol is maintained through an interconnection network between the partitions in order to maintain accuracy and reliability of data and avoid replications of them. But the question is : is there an efficient way to partition the shared cache in order to achieve an optimized performance in the system with minimum latencies and without loading the system with additional traffic in the interconnection network? And can all these be done in a dynamic way that benefits from the varying attributes of threads while their execution pattern evolves? The answers to these questions are not unique and a hasty attempt to meet to the challenges presented can be to no avail.

2.4 Conclusions

What should be noted before we proceed to the details of cache management is that its significance is proven more intensely when considering CMP and large scale systems where the outermost utilization of available resources is crucial for the optimization of the system's performance. On the other hand, the additional cost

coming from these schemes must always be evaluated before adopting such a scheme to avoid overheads – both computational and hardware – that would compromise the system's functionality. Therefore the goal in the area of cache management is focused in the best possible utilization of the shared cache at the minimum possible cost. Ways to address this issue will be presented in Chapters 4 through 7.

Chapter 3

The Use of Simflex

"Defer no time, delays have
dangerous ends."
William Shakespeare

3.1 Overview of the use of the High-level Simulator

The use of simulators for the study of architecture innovations having started a couple of decades ago, has prevailed over the past years the research in this field. However, although the use of simulators simplifies to a great extent the work of computer architects it comes with a cost. All simulators and especially full-system simulators issue a large overhead in the system inducing long latencies in the execution of applications. And while the applications we try to simulate are relatively short, these latencies although disturbing were bearable. Since however, we passed to the simulation of multicore and large scale systems running time consuming applications this overhead became much too serious to ignore, often exceeding a one million slowdown between simulator and actual hardware. Therefore, a different approach had to be proposed.

Simflex, a full system simulator running on top of simics has a different approach to this subject. Rather than monitoring all architectural details at every moment it uses sampling techniques to reduce the simulation time while maintaining the representation of the computer system reliable.

In this section we present an overview of the functionality of Simflex, of the sampling techniques used and of the reliability it offers to the simulated system. In the following section there will be a quick presentation of the various components that

compose Simflex and of the way the memory system and power management components are implemented.

Simflex, developed by the Computer Architecture Group of Carnegie Mellon University consists of a high-level abstraction simulator based on simics full-system simulator and expanding its simulating properties by achieving much shorter simulation times, an attribute particularly beneficial for large applications. The extent to which Simflex reduces the simulation time is by at least 4 orders of magnitude (~10.000) since benchmarks that used to require days to execute with the use of Simflex demand no more than a few minutes to complete. Such reduction in simulation time – especially since it comes with few drawbacks in simulation results – is easily characterized as extremely advantageous for the study of architectures when the target application is computing intensive and resource consuming. The alternative of simulations demanding long simulation times to complete has set a boundary to the extent of research in this area. The use of Simflex deals with this problem.

Apart from minimizing simulation time there are additional reasons that point to the use of high-level abstraction simulators. The process of benchmarking commercial applications demands a full system simulation including peripherals and operating system code. Finally, because conventional simulators have been optimized for high performance they are often not organized in components and are therefore not easily modified.

Simflex, on the other hand was developed to offer fast, accurate and flexible evaluation of the performance of both uniprocessor and multiprocessor systems. Its functionality is based on sampling techniques of the portions of an application that exhibit the greatest interest from the point of their impact on the performance of the overall system, rather than detailed monitoring of all architectural instances of a

system. For that reason, it takes advantage of reusable checkpoints of the state of the system. Through the encompassing of that technique can the simulator reduce the required simulation time by 4 orders of magnitude.

3.2 How it works

Simulation prior to Flexus relied mostly on detailed measurements of approximately 1 billion instructions per workload resulting in a slow mechanism, where sometimes small reductions in time came from simulating a scaled or a subset or the actual application inducing divergences from the actual workload's performance. Instead, Simflex relies on statistical sampling of the applications' performance by making many small measurements at uniform or random locations. The use of checkpoints, i.e. stored representations of the state of components (e.g. cache), enable parallel simulation of these measurements, online results and more speed for the overall simulation.

Checkpoints, however demand detailed simulation of the system, which issues additional latencies. Representative of these delays is the fact that detailed simulation runs at 1KIPS (Kilo Instructions Per Second) while functional simulation runs at 10MIPS. To avoid such cut-backs in speed, Simflex takes the abstraction level one step forward by initiating functional checkpointing of the system. *Phases*, as these representations are titled are instantiated at suitable locations so that an application's execution has approximately 2 to 9 phases and between two consecutive phases, approximately 25 *flexpoints* are created. *Flexpoints*, is the tradename of Flexus for *Checkpoints*. The purpose of these flexpoints is further explained here. Phases, store the snapshot of the system, excluding the contents of caches and branch predictors, to make a more "light" but consice representation of the simulated system. They store

the state of the system, i.e. the contents of caches and branch predictors. The early stages of each phase i.e. the first few flexpoints are used for cache warming up and training the predictors.

Deciding on the sampling periods is a crucial part of the simulation process when using Flexus since it entails the reliability of the simulation results. Both a theoritical and a practical approach in this decision take place, to find the correct balance between the sampling theorem and the special execution features for each application. We must note here that the sampling parameters are contigent upon each application and not uniform among different workloads. This guarantees a more customized behavior from the simulator, a behavior whichs results to more sound outcomes.

First of all, it is clear that sampling one instruction length execution units makes it impossible to determine CPI, therefore larger units of sampled instructions are necessary to define the localized and overall CPI of the application's execution. On the other hand, these units must remain short enough for the sampling to have meaning i.e. for the number of the total measured instructions to remain constrained. The typical size of these "running sections" is 50,000 instructions.

From the aggregate of these sampled units not all instructions are used to determine the CPI and the rest of the system's metrics. Since there is no detailed simulation of all the benchmark's instructions various execution units are forced to simulate in fast-forward execution and collect measuremets at execution units locations. These "cold" state instantiations of the system at the beginning of the execution units result in a non-random error (bias) which increases as measurement periods become shorter. In order for that problem to be alleviated a detailed – though short – period of warmup is necessary before each measurement unit to make the simulation results reliable. The extent of that warmup phase deviates among different

applications and although ROB and pipeline registers' warmup is short and predictable the same does not apply for caches and branch predictors. The latter consists of the 99% of the overall simulation time thus leading to the seaking of a different approach. In order to further minimize simulation time coming from functional warmup Simflex introduces *checkpoints* to replace functional warming which offer parallelization of various execution units, an accurate state of the system and unlike functional warmup can be proven *reusable*. The exact number of checkpoints used between two phases depends on the type of the application simulated with SPEC benchmarks ranging from 20-25 checkpoints per phase corresponding to approximately 2000 instructions for warming up the caches and branch predictors while if the case is a Server benchmark warmup units are less predictable and their determination relies primarily on empirical decisions.

Summing up the main philosophy behind the functionality of Simflex we must remind that simulation process is sampled with phases, where each phase is separated by the next one with a number of approximately 25 flexpoints conducting the warmup of various components and the functional simulation of the workload. Each flexpoint is run for a few "regions" consisting of 50,000 instructions a number of which is used for the warming up of caches and training of branch predictors and the rest for the actual simulation of the executing application. Since the first flexpoints of each phase are "tainted" from training the architecture's components they are not used to extract simulation results, while the rest of them are. Furthermore, in order to avoid gaps in simulation process flexpoints of consecutive phases overlap therefore no missevaluation of simulation results exist. A summary of this terminology and the corresponding graphical representation is shown in the following figure.

Points A, B, C and D represent phases, while between them flexpoints are instantiated (points 1, 2, 3, 4 in the figure). The initial steps of each flexpoints are used for functional warmup and the rest of them for collecting results.

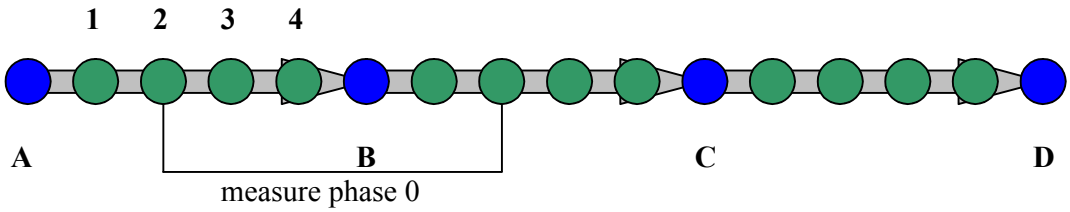

Figure 3.1: Representation of Simflex's functionality

3.3 The Simflex Modules

Flexus is a family of component-based C++ computer architecture simulators that build on Virtutech Simics's MicroArchitecture Interface (MAI) to enable full-system timing-accurate simulation of uni- and multiprocessor systems running unmodified commercial applications and operating systems.

Flexus encompasses both a simulation infrastructure and default simulation models. A simulator is composed of individual modules that are hooked together during compilation. A module is often the equivalent of a single hardware structure—for example, a branch predictor or a cache. A key strength of Flexus is its isolation of components: one implementation of a particular module can be swapped for a different implementation without requiring changes to any other modules. This flexibility also allows a particular simulator to be tailored to the needs of a specific research hypothesis. If memory system performance is being evaluated, a simple bandwidth-based processor pipeline might be sufficient. Conversely, a study that closely examines microarchitecture could use a simple memory system model. The

Flexus core provides services, such as scheduling and statistics, which are common and useful to all simulators.

Virtutech Simics enables full-system simulation. Simics is a functional simulator that allows unmodified commercial operating systems and applications to boot and run. Flexus can hook into Simics and see the instruction stream that a real system would execute. Flexus can also control Simics's timing, so as to model out-of-order effects and speculative techniques.

Flexus is designed to support the simulation sampling and checkpointing methodologies developed by the SimFlex research project. The keys to this support are flexpoints, checkpoints that store the snapshots of the state of Flexus components alongside Simics checkpoints of programmer-visible state, and stat-manager, a tool for processing Flexus statistics output to create reports and compute statistical confidence in results.

Flexus's component-based design enables easy creation of several components that all model the same hardware at various levels of timing fidelity. These components all share the same format for storing state in flexpoints. In this way, a simple, fast simulator can rapidly construct a flexpoint library, which can then be measured using a more detailed simulator. For example, the Flexus TraceFlex simulator creates flexpoints that can be used for detailed out-of-order timing simulation in a uniprocessor system with UniFlex.OoO.

Further down, we present the various components that compose Flexus aside the memory organization which will be the scope of the next section:

- **Bus/CmpBus**: Implements the bus to connect the components of each core. In the same way, CmpBus is used to implement the Bus that connects the different processors of the CMP system.

- **Cache/CmpCache**: These components will be further discussed in the following section.
- **FastCache/FastCmpCache**: These components have the same functionality with Cache and CmpCache. The only difference between them is that the former instantiates a fast implementation of the memory system where no timing simulation is possible (only trace) while the latter implements the detailed cache organization permiting accurate timing simulation.
- **PowerTracker**: (This component will be presented in Chapter 9 where power simulation is discussed since it does not consist a default component of Simflex basic edition). Its purpose is to monitor the power consumption of the CMP system.
- **Common**: This module comprises various components related to the tracking of misses or the various messages exchanged during a memory transaction.
- **BPWarm**: A module used to train the branch predictors, activated only at flexpoint execution.
- **uArch**: This module describes the micro architecture behind the CMP system, however it has not been used in any of the simulations conducted in the present work.
- **NetShim**: This component implements the interconnection network that connects the various processors and various synergestic components of it.
- **ProtocolEngine**: This module implements the coherence and consistency protocol among the competing cores of the CMP system.

Different types of simulated systems (Flexus Simulators):

- *UniFlex (/.OOO)*: This simulator corresponds to a uniprocessor system with separate IL1/DL1 caches and a L2 cache. It supports both inorder and out of

order execution.

- *CMPFlex(/.OOO)*: This simulator is used for a CMP system consisting of many (up to 32 cores) with private IL1/DL1 caches and a shared L2 cache. The various cores are connected through an interconnection network, with different topologies.
- *DSMFlex(/.OOO)*: This simulator resembles the CMPFlex simulator with the only difference that instead of a shared memory, now the L2 cache is distributed among competing cores.
- *TraceFlex/TraceCMPFlex*: Not a typical system simulator, primarily used for the instantiation of flexpoints and phases in the application's execution.

3.4 The Memory (Cache) Organization

Dependent on the simulator chosen at every case, memory organization can consist of one of more hierarcy levels. The most common case, and the one used in all simulations presented in the present work, implements private Instruction Level 1 (IL1) and Data Level 1 (DL1) caches for each core in case of a multiprocessor system, or for the one core consisting a uniprocessor system and a shared L2 cache among the cores of the CMP system. An alternate implementation targeting distributed systems is also possible, where L1 caches remain private for each core and the shared L2 cache is distributed among the cores of the system. Essentially the difference in implementation leys in different latencies and different attributes for the interconnection network and the scheduling parameters of the system. Systems, incorporating more than a L2 cache are not prone to study in the current version of Flexus. Also no systems having just IL1/DL1 caches or unified L1 caches are supported. For all memory systems there is support for ooo execution on top of the inorder version of the simulators.

As for the CMP system's memory organization, the L2 cache is shared and equally accessible to all cores and no specific sharing or partitioning scheme is by default implemented for its management.

The default replacement policies used in Flexus are :

- lru: where the victim block is chosen among the blocks in a set, as the one that resides in the cache the longest. This policy, is quite efficient in some cases but suffers from applications accessing – but not re-utilizing – massive chunks of data.
- random: in this case a random block is chosen from the set. As any other random function it bears no computational overhead but its efficiency is unreliable since the block evicted might or not be useful for the instructions to follow.
- external: this policy though not included in the default version of Simflex coresponds to the user defining explicitly which block should be evicted from the cache. Its efficiency has not been tested in the present work.

3.5 The Workloads Used

For the evaluation of the examined policies in Simflex we use a series of web and server applications running in a system of 16 cores. The main applications used are from the OLTP and DSS series of workloads and more specifically they are:

From the OLTP series the following multithreaded workloads are used : *oracle_16cpu_16cl.* As for the DSS benchmarks we use: *db2v8_tpch_qry17_16cpu, db2v8_tpch_qry2_16cpu and db2v8_tpch_qry1_16cpu.*

For the evaluation of server applications behavior we use the SPECweb99 benchmark on *Apache HTTP Server v2.0* and *Zeus Web Server v4.3*. Those workloads present different sharing and reuse characteristics to evaluate the efficiency of the proposed scheme in different workloads' characteristics.

From the workloads above, Web Server and OLTP benchmarks have high reuse rates and a relatively high level of sharing, while DSS workloads have very low reuse (resemble streaming applications) and a high-level of sharing.

The results of these experiments are presented in the corresponding Chapters, though it is safe to say here that the benchmarks that are expected to benefit more from the use of cache management algorithms are those that exhibit higher reuse and conflict rates, while those with low rates should not greatly be affected by the use of these schemes.

Chapter 4

Cache Partitioning Vol. 0

Where there is memory there is no
end (of trouble..)
Anonymous

4.1 Overview

As efficient memory utilization consists in CMPs one of the most challenging issues for computer architects, partitioning the last level shared cache (in our case L2) will be the first type of resource that we will focus on as part of our work in the present thesis. However, not all systems use partitioned L2 caches.

First of all, we must present a conventional way to address the issue of shared cache space, as is implemented by default by the Simflex simulator. According to this model, the L2 cache is considered to be shared among all threads, not partitioned and each thread has the same privileges in accessing it as any other, relying on a demand-based policy. The replacement policy used is (the default policy of Simflex):**LRU**. Simple though it may be, this replacement policy is uniform across competing threads, thus it does not take into account the special features of each executing application, not providing customized response to its needs. As we will see later the cache partitioning policies i.e. the replacement policies presented in the following chapters provide much more adapted characteristics to the needs of the processes executing and are therefore (excluding the computational & hardware overheads) much more efficient as far as cache management is concerned. We note here that by cache partitioning we mean the decision making on the block that will be evicted from the cache based on the ways that are appointed to each thread, which is why

cache partitioning is essentially a way to change the replacement policy of a shared L2 cache. These ways are decided based on execution characteristics and are not private for each thread, meaning that no duplications of data exist in the cache. If that were the case, the cache would correspond to an equal sized less associative per core cache with the pros and cons (faster search, higher miss rate) that this is accompanied with. This is the case, when we map specific ways to each thread and limit the search for the requested block to these (a cache of variable associativity). Returning to the default case of cache management implemented in Flexus, we note that no such algorithm is used and that the memory space is common to all threads and shared in a demand-based policy.

Other replacement policies (random, external and fifo) are also implemented in Flexus but the default study has been conducted for LRU and they are not expected to present better features for most of the executing applications. All these policies do not take into consideration specific features of executing applications and treat all threads as uniform, deciding on the evicted block as if there were a uniprocessor system.

4.2 Results

The results from applying the default policy to a system supporting 16 cores are presented here. The benchmarks used to evaluate the efficiency of this algorithm are web and server benchmarks (*Apache HTTP Server v2.0* and *Zeus Web Server v4.3*) from the SPEC*Web*99 suite and OLTP and DSS workloads, as are the benchmarks used in every simulation of the present work. In order, for coherence to exist between the comparison of different memory sharing policies the same benchmarks used here are the ones used in every simulation from this point forth.

We use two different types of multithreaded workloads (both from the same suite), one whose benchmarks have a high level of sharing (OLTP) i.e. communication misses from DMA accesses and OS scheduling, and the other where sharing is limited (DSS). On top of that, the first family of workloads has a high level of reuse (conflict misses) while the second does not. First of all, we present the IPC-L2 size graphs for each application when they are executed in a uniprocessor system. This is done to demonstrate the different needs in cache size these applications present when no sharing of the cache among different cores issues exist.

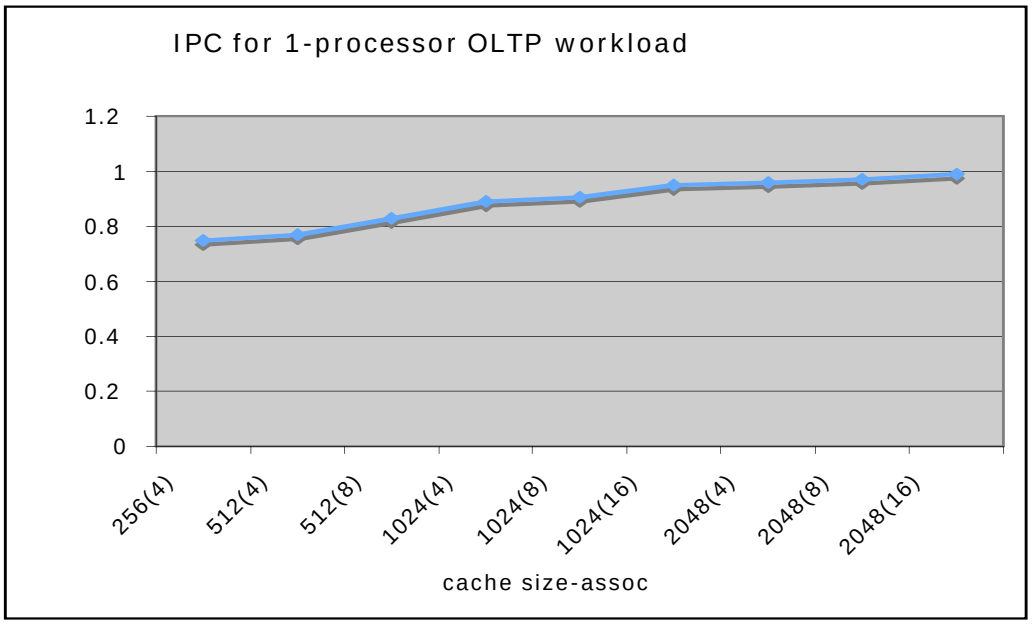

Figure 4.1: IPC for oracle_1cpu_16cl in a 1-core system.

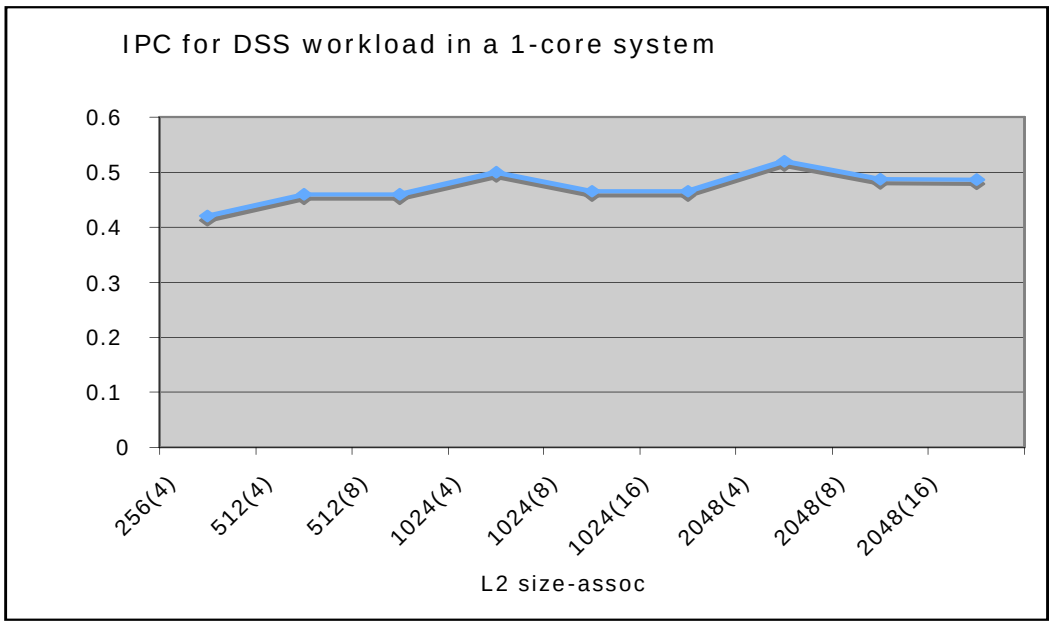

Figure 4.2: IPC for db2v8_tpch_qry17_1cpu in a 1-core system.

We see that in both cases the increase in cache size leads to an improvement in IPC, more obvious in case of DSS rather than OLTP. On the other hand, we see that since OLTP presents with high reuse rate the increase in associativity leads to improved IPC. The same does not apply for DSS where caches with higher associativity do not improve the performance of the system. This is a demonstration of the clear difference between the utilization each workload has in the cache.

Now in order to see the different types of misses in each case we maintain the cache size constant at 4MB and monitor the different types of misses in the system. Those are *coherence*, *compulsory* and *conflict/replacement* misses:

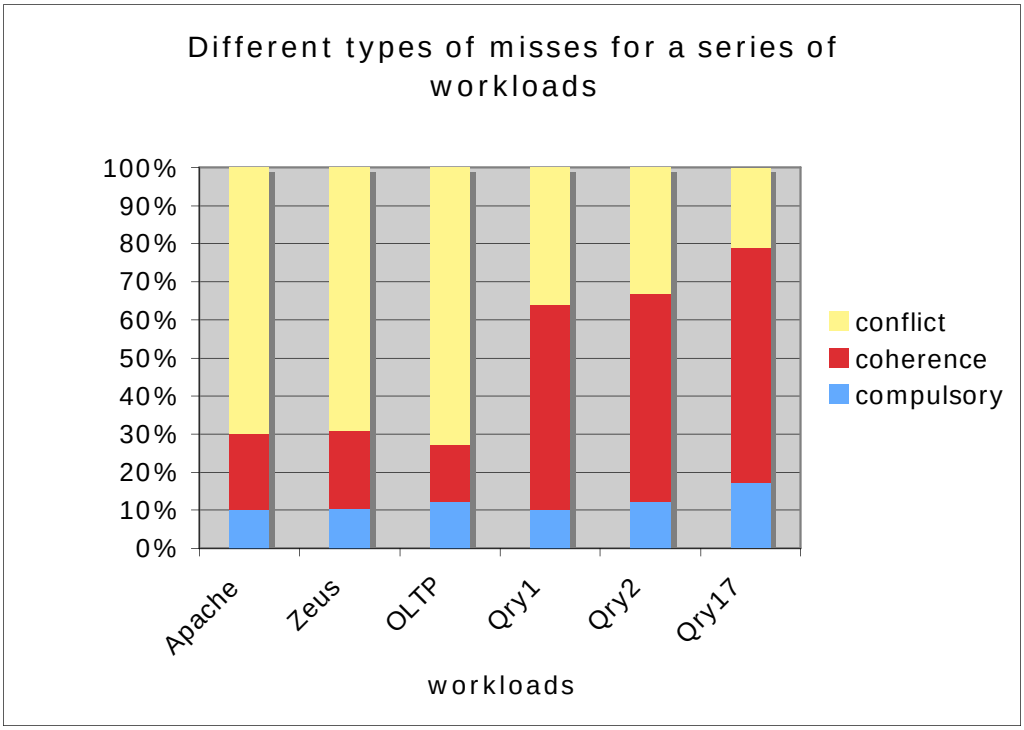

Figure 4.3: Different types of misses for Web Server, DSS and OLTP workloads.

Just as we noted before, the OLTP workloads present high reuse levels and some sharing between cores, while the DSS (Qry1, Qry2, Qry17) workloads present very low reuse (coming from the fast that they scan large tuples – matrices that exceed the cache size to extract information) and a relatively high level of sharing between cores (concurrent access in the database). Sharing however, is not the main attribute that we must take into consideration when discussing cache partitioning.

Shared data can define the uniformity of cache ways per core but not to a large extent whether the algorithm used will benefit the application greatly or not. When this is the question to be answered another feature must be placed under examination. Reuse rates, high and low demonstrate the utilization level an application has in the shared cache. Applications with high reuse rates such as OLTP in our example have many replacements because of conflict misses, therefore a more careful and customized management of the cache could help them to achieve a better

performance. The same does not apply for applications with low reuse like DSS where the main request is that of larger cache space rather than more appropriate use of the available cache. This comes from the fact that these applications will not greatly benefit from a better sharing of cache ways, since their access pattern mainly relies on the scan of large memory fragments only once and usually fragments that exceed the capacity of the cache which is why they present with high capacity misses as well.

In the following graphs we present the results from the simulation of these workloads for a 16 core system using the LRU replacement policy.

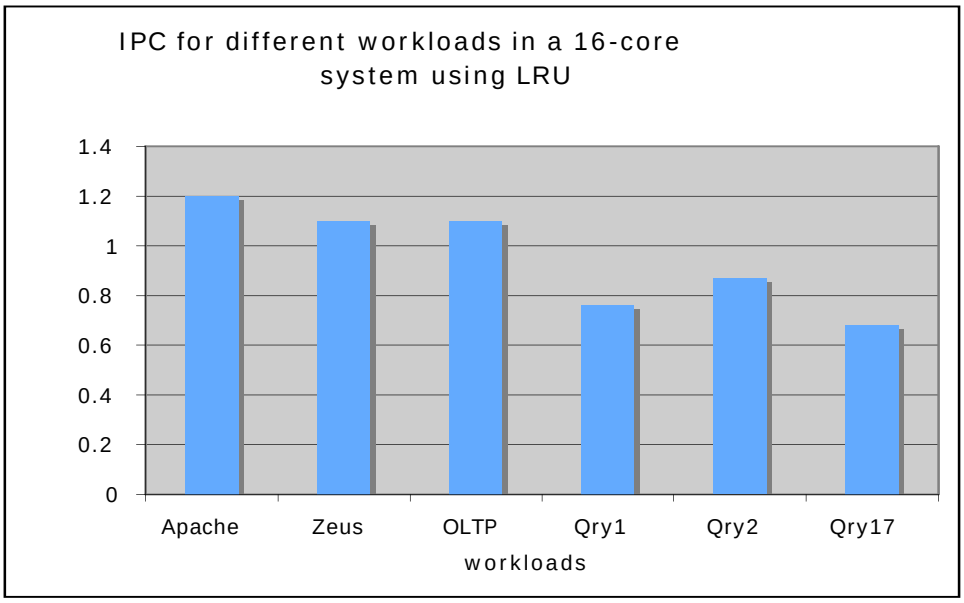

Figure 4.4: IPC for a 16-processor system with 4MB-32ways L2 cache, among competing applications for different workloads, from the web server, OLTP and DSS suites.

As can be derived from this graph, the shared memory scheme behaves acceptably for sime of the cases we tested, but there is room for improvement in most of them. While web server workloads and OLTP workloads there is an acceptable value for IPC, the same can not be said for DSS workloads. This is a result of the different types needs different applications present in the cache as was described

above. Web Server and OLTP workloads present high reuse - and less importantly sharing – levels which although not promoted by LRU are also not harmed by its use. For DSS reuse is low and that leads to many capacity misses, which LRU is unable to face, while the high sharing level that these workloads come with leads to long latencies from communication on the bus (DMA transfers) or OS scheduling issues.

This leads to a poorer performance for DSS in relation to OLTP, though a more detailed analysis of the specific features of each of the applications is necessary to understand the exact reasons that lead to this result. Therefore, though LRU is not an optimal solution for the decision of evicted blocks in a shared cache, i.e. it does not improve the performance of high reuse benchmarks, the fact that it does not help solve the capacity misses of the DSS benchmarks is what really hurts their performance leading to the extracted results. So LRU does not help any of the two types of workloads but it helps "less" the ones that could use a better utilization of the available cache space. The misses are not changed – and the conflict misses might actually be more, it is the effect of these misses however what makes the difference in the overall performance, the fact is that this replacement policy does not assist either conflict or capacity misses, but the latter result in higher latencies – and through them a lower performance – for the system.

4.3 Pros/Cons

Obviously the algorithm presented in the previous section does not relate to cache partitioning in the sense that blocks are not considered as "owned" by a specific core and therefore the decision on the evicted block is uniform across the processors of the system, i.e. no information on their execution patterns is collected and exploited. This, being a straight forward mechanism does not issue any additional

overheads in the system, however especially in high-sharing high-reuse applications it can not be considered as an efficient way to manage the shared cache. This, comes primarily from the fact that the LRU policy decides only based on the order of memory transactions and not on the benefit or even the needs of each core from the shared cache. The *key-point* in cache management is the shift in perspective from considering all cores as uniform and deciding at the level of the whole system, to considering them as independent threads with different needs in the cache and deciding on the evicted block on a per-processor basis.

Since execution patterns change in time, the desired way to address the cache partitioning issue would be by dynamically partitioning the cache among competing threads. However, to slowly examine the different steps of this procedure we first issue a simple static partitioning algorithm that appoints equally ways in the threads.

4.4 A small improvement to the default case

As we noted before, we will now try a static partitioning of the shared cache where cache ways are equally divided among competing cores. This, again does not exploit information on the threads' execution pattern and specific needs, but it will give us the chance to see how appointing a particular number of ways per core affects the system's performance. The main philosophy in this simple scheme is that every block has a coreID noting the core that initiated the memory transaction and this coreID is used to determine the current number of ways that a processor occupies in the cache. In case this number exceeds the number of ways appointed to this (and every other) processor the block evicted from the cache blong to him, otherwise a block belonging to another processor is chosen. The issue of whose processor the

block we choose is examined in a later section and will not be the subject of further discussion here. We can note however, that the differences are not that important. The number of ways appointed to each core does not change during the applications' execution.

Here we present the results from simulations performed for this scheme and both the OLTP and DSS web workloads. The simulated system has 16 homogeneous cores. The performance speedup is calculated over the LRU IPC.

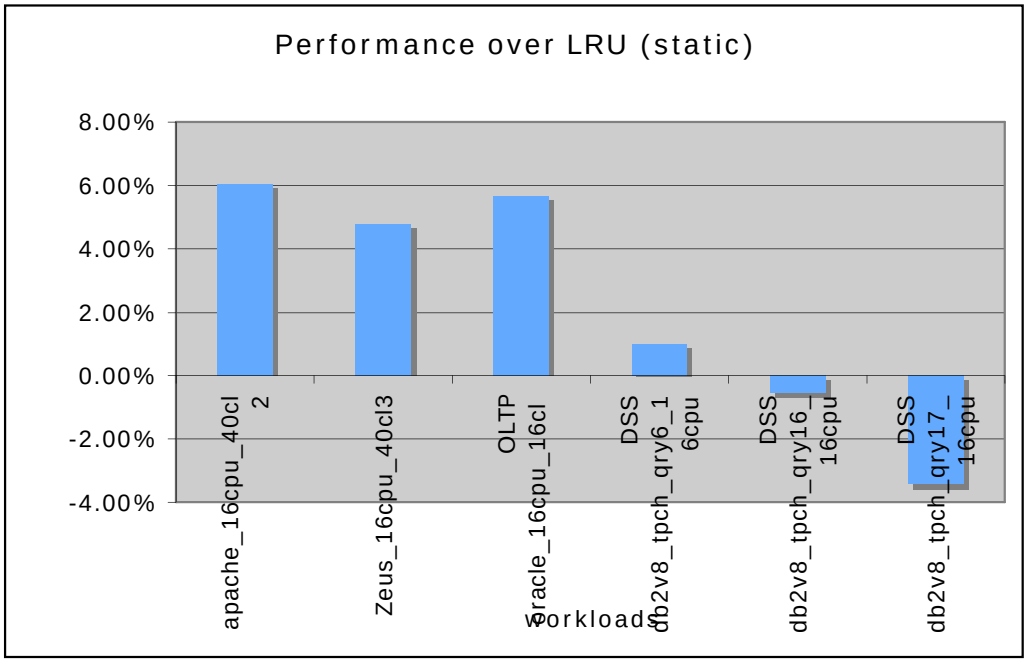

Figure 4.5: Performance over LRU for a 16-core system with a 4MB-32way L2 cache (static).

We notice that in some of the cases, there is a small improvement in performance is relation to LRU while in others there is no speedup, and in some cases there is even a slowdown comparing to the default case. This has to do, with the special features of each application which as we noted above are not taken into consideration while deciding on the cache partitions. According to this, applications –

like OLTP – which exhibit high reuse rates benefit more from the cache partitioning, even in its static form, while others that do not present that many conflict misses do not have a similar speedup. Moreover, benchmarks with a high-level of sharing such as the OLTP workloads benefit from the cache partitioning scheme more than those which do not share that many data and here in particular they present a higher speedup than they will for some of the other cache partitioning policies examined.

This has to do with the fact that in highly shared data applications the accesses are almost uniform from each core, which to a great deal is the same as in the static partitioning with equal partitions, therefore the system benefits from the partitioning scheme by maintaining the most reusable data among all the cores and uniformly sharing the cache between synergestic – here – processors.

A small improvement to that would be a scheme with static allocation of ways, but unlike in the one presented here, the ways are not equally partitioned but rather according to the needs of each thread, thus adjusting to the features of each application. This deals with the problem of different threads presenting different execution patterns but not with the fact that each thread presents different patterns during the completion of its execution unit.

The algorithms presented in the next chapters of this work, do not use static but rather readjust dynamically the cache partitions among the competing threads. The results from these simulations are presented in the related sections.

As a last note for this chapter we make a comment on the effect of partitioning algorithms on various types of workloads, results which will be confirmed by the experimental results in the Chapters that follow. According to this the following remarks are true about the workloads used to test the proposed schemes.

In case of a high reuse application (conflict misses) it is safe to say that cache partitioning will be of aid since it will increase the appointed ways where there is

need. Now, as far as sharing is concerned what we can say without delving into the details of each application relates more to the pattern in which the ways will be shared rather than to the extent in which it will improve the system's performance. In high level of sharing it will be more uniform (not many misses – though some longer latencies from communication therefore no radical changes in the partitioning will be necessary) as for a lower level of sharing it becomes more unpredictable the pattern of the appointed ways after the implementation of the partitioning algorithm. In either case, the applied policy is expected to adjust to the current needs of the competing threads. It is expected for high-sharing applications such as OLTP workloads to present a better speedup in performance than those with a lower sharing The same applies for applications with high reuse which will find the proposed schemes a lot more beneficial than applications where the cached data are only accessed once (and potentially the data range is larger than the available cache space leading to high capacity miss rates).

Chapter 5

Cache Partitioning Policy Vol. 1

"Don't fear failure so much that you refuse to try new things. The saddest summary of a life contains three descriptions: could have, might have, and should have."
Louis E. Boone

5.1 Overview: Cache Partitioning Algorithm using a Bloom Filter

Presented in the previous chapter, we saw the results of a completely shared L2 cache between the competing applications and evaluated its impact in the overall system. The replacement policy used in the "default" LRU scheme has been proven quite efficient and low-cost for most applications, and is therefore used in a large portion of uniprocessor systems. The situation in multiprocessor systems though varies. Here, we take the step forward to actually manage the L2 cache by partitioning it among the executing tasks at any given time. This, being a competitive effort has many potential approaches, largely differentiating on whether the partitioning algorithm relies on microarchitectural or parameters of a higher level.

First of all, we need to specify the steps needed for a partitioning algorithm. In the last section of the previous Chapter we analysed a static partitioning scheme, where however there was no other functionality other than the stragiht-forward partitioning of the ways among competing applications. Here we will see that for the algorithm to have the effect that we want additional steps are necessary. In order for a partitioning policy to be applied on a CMP system there are two levels that must be addressed. The first part of the algorithm relates to *monitoring* the system in order to

define the access pattern of each thread whether that is approached as inserting coreIDs to blocks in the cache or as buffers implementing a per-thread lru list. The decision on the monitoring scheme is crucial and can often be determinant of the efficiency of a cache partitioning policy. As we will see in both this and the following chapters the way a policy uses to monitor a number of tasks greatly varies from algorithm to algorithm in both hardware and computational overhead as well as in the efficiency of the final scheme. Secondly, the data received periodically from the monitoring process are evaluated by the *partitioning algorithm* in order to determine the new partitions appointed to each thread. This consists the second part of the partitioning algorithm and can also be addressed in many different ways – utilization level, search algorithms - however they all share the concept of feedback, i.e. the idea of collecting the results provided by monitoring and redistributing the cache between the different threads based on the current statistics of their execution (current access pattern, level of utilization of given memory space).

One of the most important defects of previously used algorithms to share a LLC – the static partitioning that we attempted included - is the lack of taking into consideration the special features of each program or on the other end the overuse of such statistical measures in an attempt to achieve memory management of higher efficiency leading to high – and therefore not-wanted – hardware overhead for the system. The algorithm presented here achieves a balance between the two via using information on execution patterns coming from a monitoring process during the execution of the program and escaping the trap of overloading the system with sampling fields that add a large hardware overhead to it.

As far as this last point is considered an optimization is proposed in the last section of this chapter, one that leads to a partitioning algorithm of the same efficiency but doesn't burden the system with a hardware overhead this heavy.

In the following section we describe the partitioning algorithm based on a Bloom Filter. First of all, we demonstrate the mechanism used for the monitoring part of the policy:

As we mentioned before, the LRU policy, easy in implementing though it might be does not evaluate the different features of executing applications in a multicore system and therefore does not utilize effectively the shared LLC. The scheme presented here offers a better utilization of the cache space by dynamically allocating the memory resources based on the level that each application will benefit from them. The main goal is to allow applications with varying cache utility profiles to all benefit from the shared cache. This means that the ways of the L2 cache will be partitioned among high, low and saturating utility profile applications in a dynamic way that adjusts to the current behavior of the program. In order to do so, an additional circuitry is needed to perform the monitoring of various attributes for each of the applications and maintain the metrics that are necessary to determine the current partitions. As we will see in the results of the experimental methodology the gain from using this scheme resembles an increase in the size of the L2 cache of about 50% with an actual additional hardware overhead of 0.55%.

Let us first explain the initiative behind the implementation of this scheme. Preceding cache partitioning schemes have benefited either from a per-set fine grained monitoring of the applications' access pattern or by a more coarse-grained granularity monitoring method that unifies sets' statistics to achieve a more low cost scheme. Both ways of addressing this matter have advantages and disadvantages. In this policy we try to use a per-set monitoring method to collect traces on the applications' execution pattern and utilize these information to determine the desired ways per application in the shared cache. This scheme performs an effort not to add a very large hardware overhead to the system, but at the same time maintain accurate

per set metrics on thread's execution. Future modifications of this scheme turn to OS-performed thread scheduling based on the information derived from the LLC promoting either fairness or priorities among competing threads.

The policy presented here is based on the functionality of Bloom Filters. Bloom Filters consisting of probabilistic structures to maintain set membership offer a very compact representation for a small trade-off of some false positives. The main idea is to use the Bloom Filter to represent misses that would have been avoided had the thread been appointed more ways in the L2 cache.

In order to do so, one BF per core per set is maintained and for tagging purposes the last k bits of the address's tag are used. Moreover, to do this correlation between sets and BFs we need to know the processor that has issued the access of each set in the LLC. To do so, a coreID is maintained in each cache line demonstrating the core that initiated the memory transaction. The partitioning algorithm decides on the new partitions based on hit/miss rates for each application. However, not all miss tracking is necessary. The misses that interest us are only those that would have been hits had the application been appointed more ways in the cache. Those – described as "far misses" – are the ones tracked by the BFA and the ones that determine the decision of the partitioning algorithm.

Every time a line is evicted from the cache the *k* least significant bits of its tag are used to index a bit in the BFA which is then set. On a cache miss the BFA is searched for the appropriate tag and in case that tag is set a "far miss" has been detected. If that is the case the set bit in the BFA demostrates that the corresponding cache line would have been a hit if the application was given an additional way in the shared cache, thus affecting the decision of the partitioning algorithm.

Although Bloom Filters eliminate the possibilities for false negatives, i.e. bits in the Filter that should have been set and are instead zero, the same does not apply

for false positives since the fact that only k from the n bits of a tag's address are used in the BFA causes aliasing to appear. Ways to address this problem relate mostly to the increase of the number of bits used, though that directly leads to extensively increased hardware overhead for the system. Another way that has recently been proposed to avoid such hash collisions uses up/down counters in each hash location instead of single bits. The counters track the number of addresses hashing into a particular location. Upon miss in that address the counter at the indexed location is incremented by one and upon commit the counter is decremented by one. The counters can either be made sufficiently large so as not to overflow, or they can take some other corrective action using one of the techniques described below when they overflow. The use of counter based Bloom filters was previously proposed by Fan et al. .

Bloom Filters with more that 1-bit per entry are used, though not suggested in systems where hardware overhead is a limiting factor. They appear with different purposes. Implementing BFAs with more than 1bits in one case allows the algorithm to evaluate information about the past of the accesses for each application i.e. misses that would have been hits had the thread been appointed two extra ways. The priority in case of 2-bit filters is 11 -> 10 -> 01 -> 00. These structures are more fast in responding to changes in the access patterns of applications since they require two misses in the same set (during one iteration) to increase the ways of a thread by 2 while in a 1bit filter that would require two misses in the same set in two consecutive iterations. The issue of aliasing is again present. Another reason why 2 or more bit BFAs are used is for the implementation of up/down counters tracking the repeaing misses in a specific address as those explained previously. The difference between the two implementations is that the second one detects the frequency of misses in a specific place in the Bloom Filter i.e. a specific address in the cache while the first

one represents misses that happened in particular recency positions in the cache ways (lru, next-to-lru, thus resembling utility-based monitoring as this presented in the following Chapter). A third use of 2bit Bloom Filters is presented in Chapter 10 and is therefore not explained here extensively.

A quick note here relating to the order in which sets are evicted from the cache. No such information is preserved since bits in the Bloom Filter that are set keep no information of the previous and next addresses which resulted in a far miss. Such a monitoring is possible but as expected issues an extra overhead in the system. Furthermore, number of misses in the same set that exceed the number of cache ways can not be represented in the filter consisting of 1 bit. If such a case monitoring is essential for the system BFAs with more than 1 bits – like the ones presented above – are proposed.

Hash collisions resulting in high false positive rates can be minimized through the reevaluation of the proposed hash function, i.e. the method used for unsetting the bits, and the size of the BFP tables. In the experimental methodology section we present the use of two different hash functions, one ($H0$) common among all workloads and the other ($H1$) dependent on the current workload since it uses profiled heuristics to generate an index using the bits in the physical address that were most random on a per-benchmark basis. The first hash function does not issue any computational overhead to the system, while the second one adds the latency of one XOR-gate. To determine $H1$ for each benchmark, we populated a matrix by XORing each pair of bits of the address and adding the result to the appropriate position in the matrix. We then chose the bits that generated the most even number of zeros and ones, assuming that they were the most random. Both these schemes have been tested and the results appear in Section 5.3 .

In case, a less extensive hardware overhead is preferable we can replace the per set monitoring of the cache blocks with a more coarse-grained technique relying on sampling of the blocks to derive information on the threads' execution while issuing a hardware overhead equal to the previous one divided by the number of blocks sampled. If this is the case, attention must be paid not to choose blocks with the same k least significant bits since that would result in serious faults in the statistics for the cache. Usually the blocks sampled are chosen to abstain from each other by 64KB. Again, the exact percentage of the total number of blocks used for sampling, is up for discussion, but it has been proven [QuP06] that a relatively small percentage is representative of the total cache behavior and therefore enough for sampling purposes.

Having explained the monitoring part of the algorithm, we must now present the actual partitioning scheme. The decision making algorithm is based in the evaluation of a gain over a loss function mainly determined by the hits in the LRU position which would become misses did the processor have one less way in the cache vs. the far-misses in the LRU position, as indexed in the Bloom Filter for a particular set. We must note again here that the partitioning of the ways is done per set therefore the algorithm works in a very fine grained granularity.

The loss function is defined as the hits in the LRU position while the gain function is defined as the multiplication of the far misses with a factor $a = 1 - \frac{ways_{occupied}}{associativity}$ as shown in (1) and (2).

$$lose = C_{LRU} \qquad (1)$$

$$gain = a \cdot C_{FAR-MISSES} \qquad (2)$$

The exhaustive search algorithm will try to find the best partitions for each core (per set) and to do so, it will examine all possible combinations. After a number

of cores, which does not exceed 8-16 it becomes clear that this search is very inefficient since the computational overhead scales exponentially. Therefore, another approach had to be proposed.

A linear algorithm in place for the exhaustive one, provides a suboptimal solution for each core without leading to high computational overheads. Instead of evaluating all possible combinations of ways per core this algorithm compares the current maximum gain and minimum loss to "exchange" the extra ways between two cores. The exact steps of the algorithm are:

```
for core = 0 to N-1 do
        gain(core) = C_LRU
        lose(core) = C_FAR-MISS
sort gain, lose from max to min
while max (gain(core_i)) > min (lose(core_j)) do
        associativity_i = associativity_i + 1;
        associativity_j = associativity_j – 1;
        remove core_i, core_j;
        cores = cores - 2;
        if cores ≤ 1 then
                return associativities;
        end if
end while
return associativities;
```

Since in the worst case N/2 comparisons will be needed the complexity of this algorithm is O(N), where N the number of processors in the system. Having proven that the computational complexity increases linearly with the number of cores the proposed scheme is considered to be a relatively scalable algorithm, since neither the computational nor the hardware overhead, aside from the increase in the bits needed

for the representation of the coreID, present a significant increase relevent to the system width. We will see that this is one of the most important characteristics of ABFCP which has led us to use it when proposing an alternate scheme for improved cache partitioning.

The repartitioning period is 10000000 cycles to avoid a large computational overhead, but can be adjusted to the needs of various applications, i.e. the speed of modifications in their access pattern during execution.

For the evaluation of the algorithm presented in this section we used both multiprogrammed and multithreaded web server, OLTP and DSS workloads of similar or different attributes. The results of the experiments are presented in paragraph 5.4 .

The decision for the to-be-evicted block happens by comparing the ways that the core that initiated the memory transaction currently occupies in the cache and the maximum number of ways that it can occupy based on the latest partitioning. If this number is found to exceed the number of ways that are appointed to the core then the block that will be evicted belongs to him, while if it has more available ways to accupy another block belonging to another processor will be chosen for eviction. The choice of whose processor this block will be has been discussed previously (Section 4.4) and will not be repeated here. We will only note that we choose this block by evicting a block from the processor that has the greatest number of ways in the cache. Alternative proposals are still possible, and at times more efficient. Now as far as which one of the blocks owned by a core will be evicted at a fetch address transaction, what we have followed is choosing the one that is placed in the LRU position, by rearranging the list of blocks so that the first in line is always the LRU entry. The list is common across all cores. Here as well there can be alternate approaches, but this subject will be more extensively discussed in Chapter 10 where some of these

approaches are presented (being an area more closely related to adaptive insertion issues in the cache).

5.2 Implementation of Cache Partitioning Scheme Using Simflex

In order to test this algorithm we have implemented it using the infrastructure of Simflex simulator, whose functioning features have been presented in detail in Chapter 3. As part of the present Chapter we will present the specific modifications necessary to perform the desired experiments as well as an effort to evaluate its efficiency and additional hardware overhead in the system.

We must note here that we are looking for timing simulations therefore the modules of Simflex which we have modified are Cache and CmpCache which simulate the exact latencies from potential cache misses while if we where interested in mere statistics and a more simplified memory hierarchy we would use FastCache and FastCmpCache where specific latencies are overlooked. Furthermore, the simulator used is CMPFlex implementing a multicore system with private IL1 and DL1 caches and shared L2 cache rather than DSMFlex where the L2 is distributed both on and off chip. In both cases the cores are of the same type (therefore no heterogeneous matters are evoked) and are connected with a torus interconnection network. The type of modifications on the interconnection network are feasible in a study of its management among competing cores, but this approach exceeds the scope of this work and is not further analysed (see Chapter 11).

5.2.1 Defining the coreID

A necessary attribute of the algorithm presented in the previous section is the coreID, an extra field extending the cache line which determines which core utilizes a specific block. This extra field though correlated with a corresponding cache line does not need to physically be an extention of the actual memory, therefore although it adds to the hardware overhead of the system it does not affect the access time in the cache, thus not causing additional latencies.

The implementation of this coreID in the simulator comes by adding a variable number of bits (dependent on the processors number – system width) in the cache configuration file, here residing in the NewCacheArray.cpp file, also encompassing the implementation of the replacement policy as we will see further down.

The coreID is initialized with the coreIdX field provided by the message that is instantiated at a cache request (see Chapter 3 Simflex Modules – MessageTracker) and this id is used from that point forth as part of the partitioning policy to determine the ways that need to be appointed to each given core of the system. As we have stated in the presentation of the algorithm, this coreID since it accompanies every cache block adds to the hardware a significant overhead. In the last section of this chapter we will see a way to constraint this overhead by virtually mapping ways to specific cores. However, as far as the simulator is concerned this modification has no difference in the way the algorithm is written and only relates to the actual hardware design of the system.

5.2.2 The Monitoring Scheme/The Replacement Policy

First of all, we must note that the replacement policy used is implemented per set meaning that the victim block is picked among the blocks of the corresponding set rather than the whole cache, which would lead in must lower performance gains for the system.

The new partitioning policy added here (REPLACEMENT_BLOOM) is the one that will be further explained in the current section. We maintain an ordered list, to represent the way the blocks have been inserted in the cache (MRU to LRU) which remains common across the cores. Rearranging its blocks, we show the temporal access to them by each core (i.e. if a core accesses a block that is already in the list, it is being moved to the mru position maintaining the same coreID in the related field, while if the block belonged to another core, it is moved to the head of the list and its coreID is updated. In case the block was not in the list it is moved in the mru position and another block is evicted from the lru position of the list and that is what will be explained next). Those rearrangements in the lru list is the main modification performed in NewCacheArray.cpp file. The actual implementation of the replacement policy is contained in the NewCacheArray.hpp file. The choice on the evicted block relates to the ways that have been appointed to each core. A comparison is performed between the current ways of the processor accessing the block and the maximum of that value. If the core is already using its maximum ways, no more lines in the cache can be given to it and should therefore evict one of its own blocks to insert the new one. The block evicted is therefore the LRU block – based on the arrangement of the list – that belongs to the current processor. On the other hand, in case the processor is not using the maximum of its available ways in the cache it will insert the new block

in its MRU position and the block evicted will be that of another processor. We have chosen to evict the block of the processor with the most cache ways in the system.

The first part of the cache partitioning policy is the way the Bloom Filter and the monitoring scheme is implemented. The number of ways per core is chosen based on consulting the Bloom Filter data as explained in the previous section (5.1-*Overview*).

We define one Bloom Filter per set having 64 entries and set the corresponding bit (indexed by the 5 least significant bits of an address) when a miss is detected in this set by the processor that initiated the transaction. This miss as previously explained is called a "far-miss" and will help in the decision for the ways that will be appointed per core. This detection of far-misses is performed while the algorithm searches for the victim block. As can be seen from the corresponding code we perform a comparison to decide whose block will be evicted and then search for the actual block belonging to a specific processor. In case we have to evict the lru block owned by the processor that initiated the request when this block is found (residing in the LRU position) it is evicted from the cache and the BF entry is updated. In case we evict someone else's block we increment the current ways occupied by the processor and evict the appropriate block. Far misses are calculated by adding the contents of a specific BF corresponding to the appropriate set. The code that describes the monitoring and replacement process described here is the following:

```
int i=theAssociativity-1;
if (current_partition[coreIdX]>assoc[coreIdX])
{
        while ((coreID[i] != coreIdX) &&(i>=0))
                i--;
        int k_bits = theBlockTag && 0x3F;
        BloomFilter[theSet][k_bits] = 1;
        return theMRUOrder_partition[i];
}
else
```

```
    {
        while ((coreID[i] == coreIdX) && (i>=0))
            i--;
            current_partition[coreIdX]++;
        return theMRUOrder_partition[i];
    }

Cfar_miss += BloomFilter[theSet][k_bits];
```

Although the monitoring scheme is implemented in this file the definition of the new replacement policy is added in the corresponding *.cpp file containing all the previously implemented replacement policies.

5.2.3 The Partitioning Algorithm

As was mentioned above, the same source file contains the code to implement the actual replacement policy for the LLC therefore additional changes are needed to for it perform in the desired way.

As far as the actual partitioning algorithm is consedered throughout the execution of applications, cycle counts are calculated counted through the CycleCount() and every 10 mil cc an iteration of this algorithm is executed. Other values for the timeout have also been tested and the results are presented in the next section.

When such a moment occurs, both the loss and gain functions described in the ABFCP algorithm are reevaluated based on the new statistics for far-misses and lru hits and according to those values the new partitions are appointed to each core. Loss and gains functions are sorted from max to min and according to a linear algorithm – to minimize the computational overhead for large scale systems – ways are distributed among cores. This, though not the optimal solution is efficient enough and fast

enough to be an acceptable solution to the partitioning of the cache ways. The code used for the repartitioning is as follows:

```
if (cycleCount()-partition_period<1000000) ;
else
{//calculate new repartitioning
for (core_num = 0; core_num < system_width(); core_num++) {
   lose[core_num] = Clru[core_num];
   gain[core_num] = (1-assoc[core_num]/theAssociativity)*Cfar_miss[core_num];
   }

Lose = sort(lose, num_cores);
Gain = sort(gain, num_cores);

N1=Lose(0);
N2=Gain(0);

int core_G[N1];
int core_L[N2];

core_G=inv(Gain);
core_L=inv(Lose);

while ((Gain[0]>Lose[num_cores]) && (num_cores > 1))
{
   assoc[core_G[Gain]]++;
   assoc[core_L[Lose]]--;
   shift_left(Gain);
   num_cores=num_cores-2;
}
   partition_period = cycleCount();
}
```

With this algorithm ends the ABFCP implementation and we proceed to the presentation of the results that came from the execution of this scheme. Moreover, this is the "default" Bloom Filter implementation that will be used for the alternative schemes that are based on this and that are presented in the following Chapters.

Before we proceed to the observation of the results we must note here that from this point forth we always bind a thread to a processor in order to avoid errors in the coreIDs that are critical for the development of the partitioning algorithms presented. Therefore whenever we refer to a thread we refer to the corresponding core or

processor as well. From now on the terms cores processors, threads and programs will refer to the same thing and thus will be used with no discrimination.

5.3 Results

The results from simulating a series of multithreaded workloads in a multiprocessor system are presented in this Section. The benchmarks used are as previously Web Server, OLTP and DSS workloads which present different characteristics as far as reuse rates nad sharing is concerned.

All simulations have run for 100 million cc, which in most of the cases corresponds to 10 flexpoints per applciation.

The results from simulating a 16-core system with a 4MB, 32-way L2 cache using the ABFCP scheme with BFAs of 5-bits monitoring per-set and using the *H*0 hash function are as follows: The repartitioning period is 10 mil cc.

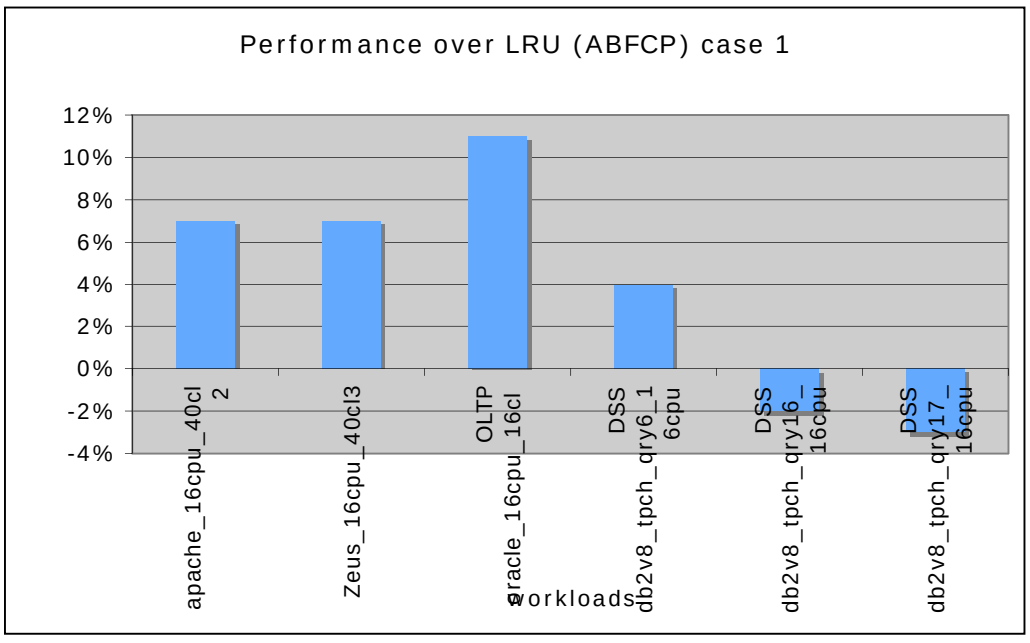

Figure 5.1: Performance over LRU for ABFCP in a 16-core system 4M-32way L2 cache

We notice that the scheme's impact on the system's performance varies among different workloads. This of course is expected if we take into consideration the different features these workloads present. As far as the Web Server (apache & zeus) benchmarks are considered we observe that the system exhibits an improrant speedup coming from the fact that these workloads have high sharing and reuse rates (database servers), which greatly benefit from a better utilization in the cache. The fact that now cache ways are more efficiently partitioned among competing cores leads to a decrease in the appearing conflict misses, and in an improvement of the overall system's performance. Same is the attitude for the OLTP workloads which also have very high reuse rates and sharing levels among the different processors.

This high reuse rate as expected is very convenient to display the functionality of the algorithm tested since this is exactly its main purpose. To distribute the available ways in a way that will adjust to the needs of each thread and permit the application to avoid conflict misses that resulted from the previously used LRU replacement policy.

The sharing level of the application does not really affect the benefit from the partitioning scheme but mostly the way the cache is distributed among cores.
Indeed as we have predicted due to the high sharing level the partitioning algorithm after a number of iterations results in distributing almost uniformly the ways across applications, since their needs are very close to each other.

As for the DSS workloads, there thet results are different. ABFCP instead of improving, hurts the performance. This can be accredited to a number of reasons, all of them related to the features of the algorithm and the executing applications. As far as the applications are concerned, the fact that their reuse rate is low has as a result that they will not benefit from the use of the cache partitioning system. It does not

however justify the lowering in performance. This unexpected result, has to be searched in the main defects of the proposed system.

As we noted in Section 5.1, the monitoring through the use of 1 bit Bloom Filters can not track multiple misses in the same address therefore all set bits in the filter are treated as with the same weight (importance), in order for that to be possible we need Bloom Filters which use counters. Furthermore, BFAs with 1 bit can not detect misses in other recency positions except for LRU which makes it difficult to determine how many additional ways are required for a thread to improve its performance. These schemes are explained in the next section. Finally, in case of executing applications with faster changes in their execution pattern than the repartitioning period we expect that they will not benefit from the cache partitioning scheme as would applications with access pattern variations that are more infrequent. That is the case for DSS qry16 and qry17 which change very quickly the cache space they access, between low and high utility, thus the partitioning algorithm has trouble tracing these changes and adjusting efficiently to them. Problems like this can be alleviated through tuning in the cache partitioning period – though the rising of computational overhead is once more not to be overlooked.

There are some additional results from variations of the parameters of the initial system. Though the number of cores and the size-associativity of the L2 cache remain the same, we modify the hash function and the number of bits in the Bloom Filter to avoid high aliasing phenomenas. The alternate hash function is $H1$ as was presented in 5.1 . The results from these simulations are as follows:

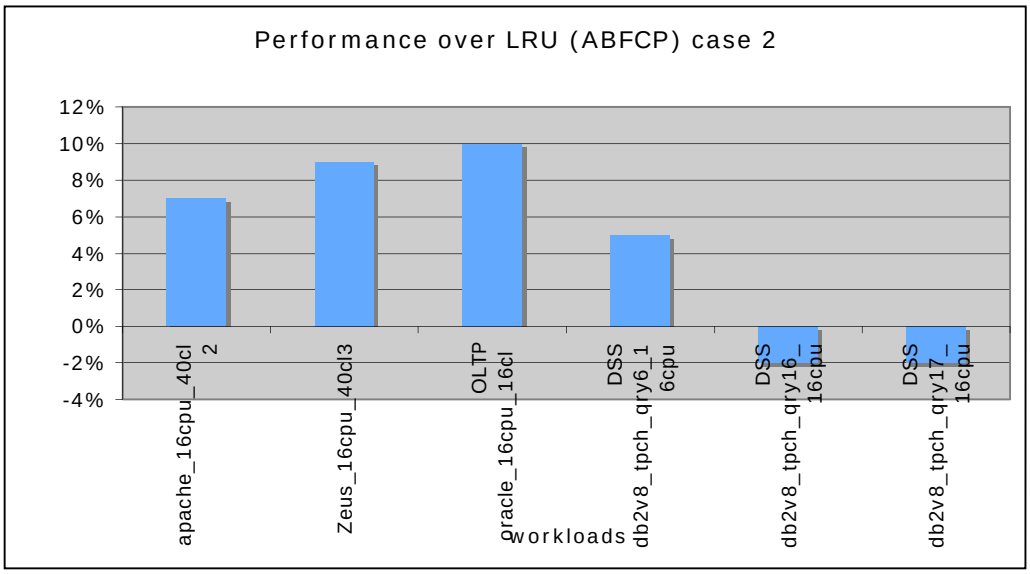

Figure 5.2: Performance over LRU in a 16-core system with a 4M-32way L2 cache (H1)

We observe that there is no significant change in the results, other than approximately ±1% variation in speedup. Had this improvement (in some cases) have come with no additional computational overhead, *H*1 might have been a good idea, but since it adds to the computations one XOR-gate per access this modification is not important enough to be adopted.

The same applies for the variation in the number of bits in a Bloom Filter. Here are the results from a 6-bit index when all the other parameters are maintained the same.

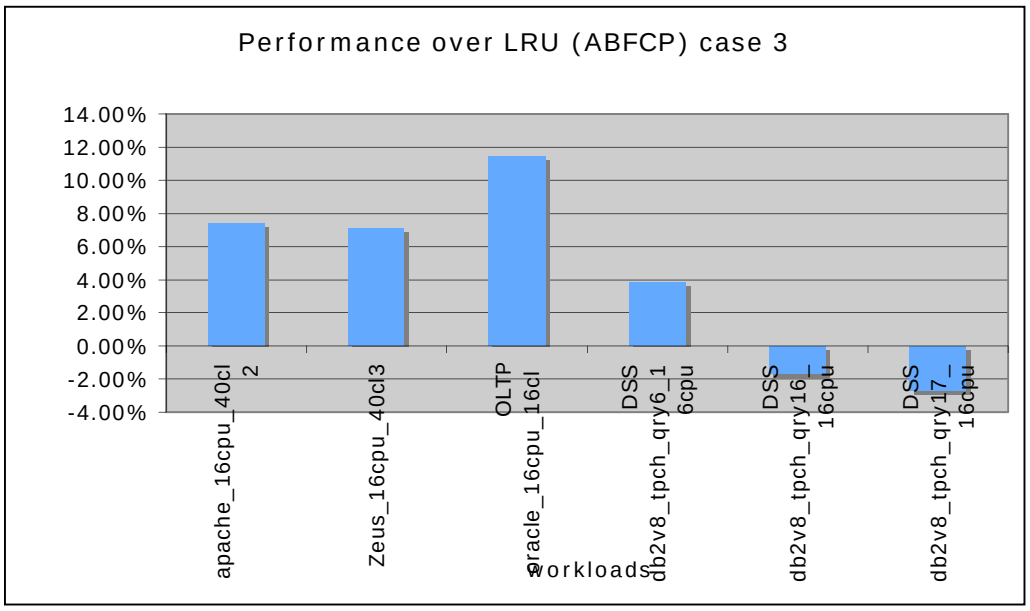

Figure 5.3: Performance over LRU in a 16-core system with 4M-32way L2 cache (6-bit BFAs).

Again the improvement in performance (coming from lower aliasing effects) is not that important compared to the additional hardware overhead from the increase in the BFAs' bits. Therefore, as before it does not worth to adopt this modification.

Finally, here is the simulation of the above workloads for a different repartitioning period (1 mil cc):

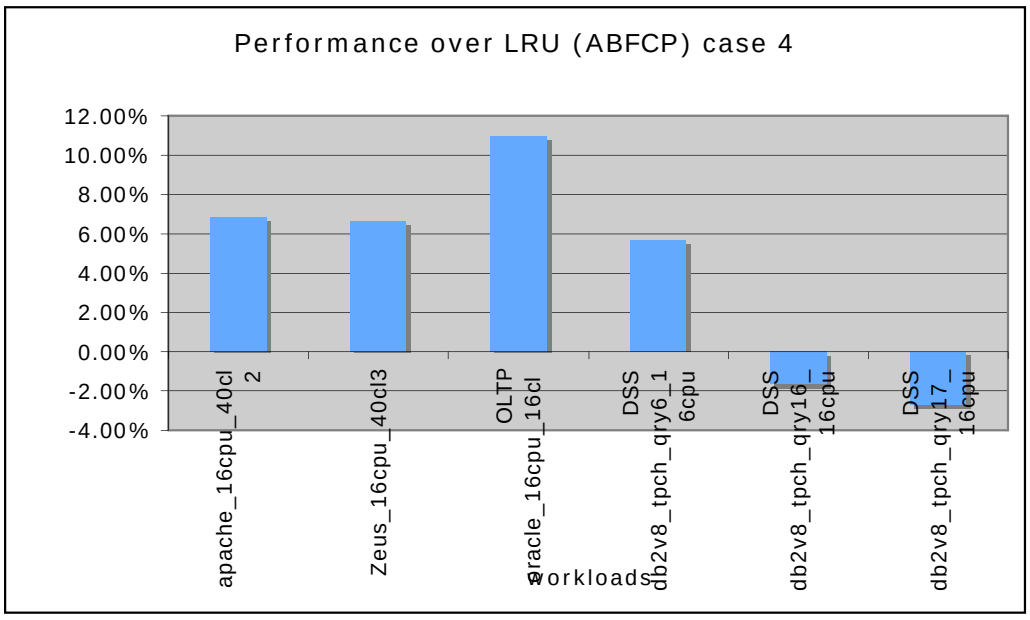

Figure 5.4: Performance over LRU in a 16-core system with 4M-32way L2 cache (1 mil cc).

Indeed, as we have noted qry16 and qry17 behave better since this new repartitioning period better fits their access pattern modifications. However, this increase in the frequency burdens the system with additional computational overhead (and potentially higher power consumption) so it should not be decided recklessly.

We notice that the differences from the modification of these parameters, though not unexisting are not very important, which leads us to believe that the features of the applications as they were described above and the main defects of the system are the ones that mainly affect the results of the performed simulations for a multiprocessor system.

5.4 Other Partitioning Schemes Based on a Bloom Filter

The algorithm presented here is not the only way of using a Bloom Filter to decide on the partitions given to each core. The filter used here has one bit information which corresponds to the block evicted from the LRU position in the

cache. Filters using more bits are also an option although the gain they offer as far as performance is concerned is not as significant as the overhead they issue in the system. A series of simulations with a filter using 2 bits for the representation of the blocks evicted from the cache (thus containing more information of the past use of the cache space) have been conducted and the results are presented in the following graphs. The second bit in the Filter represents the block that would not have been evicted from the cache had the core been appointed two extra ways. The benchmarks used are again web Server, OLTP and DSS workloads as before.

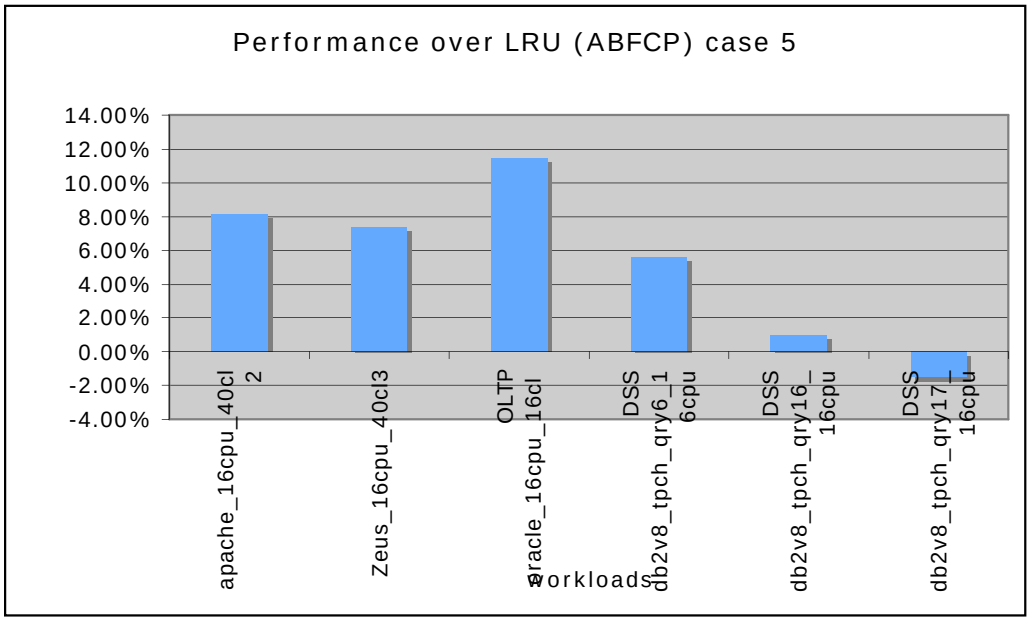

Figure 5.5: Performance over LRU in a 16-core system with 4M-32way L2 cache (2 bit BFAs).

As can be seen from the above graph the improvement from the use of 2 bit filters is significant although the additional hardware overhead (an order of 0.55% of the total size of the cache).

Apart from 2 bit Bloom Filters to represent other recency positions there can be BFAs implemented with counters like the ones described in the beginning of the current chapter. Those implementations are also likely to offer a slight improvement in the performance of the system by tracking multiple misses in the same (or in aliased) addresses, but the overhead that they issue makes them unlikely to be adopted

in systems that do not seek very detailed management of the shared cache. This taken into account the study of such schemes exceeds the purpose of the current work, although a similar design is used as part of a combined effort to manage the cache presented in Chapter 8. More details for the implementation of counter-based BFAs can be found there.

Finally, we note here a slight modification to the previously presented scheme,in order to make the hardware overhead less. According to that idea instead of keeping track of the core that initiated a specific memory transaction through the additional field of coreID which burdens the system with an overhead proportional to the size of the cache, we can virtually map certain ways of the cache to every processor. Like this, specific ways in the cache will belong to a certain processor in the sense that if for example the algorithm decided that core 1 must have 2 ways it will be appointed the first two ways in the cache. Finding a block in these ways instantly means that it belongs to core 1, alleviating the need for the coreID to exist. The partitioning algorithm remains the same but now every time we seek or want to place a block we know where in the cache to look for it, keeping track of just the ways that are given at any given moment to each processor.

The mapping of ways to cores leads to a decrease in the required hardware of about 0.1% of the total 0.55% that it initially required, phus with the extra benefit that this hardware unlike the previous one is not proportional to the size of the cache but to the number of cores, which leads to a more scalable solution for the physical design of the partitioning scheme. And although in case of per-set monitoring this difference might not be that impressive, if a more coarse grained scheme is chosen the difference in required hardware broadens making the new implementation more appealing. This comes from the fact that in that case the lines of ways per core tracked are decreased while the coreIDs per set remain the same increasing the required overhead.

To sum up the features the presented scheme presents we can say that it provides a relatively important performance improvement at a reasonable cost, though as we will see further down other schemes offer the same speedup at a lower cost. As far as scalability is concerned BFAs are scalable enough since the number of cores does not significantly affect the implementation of the algorithm other than the fact that it adds to the coreID field more bits and the fact that the value of ways per processor decreases leading to potential mistakes in the decisions. This however is not a mistake of the Bloom Filter but of the constraints in cache associativity and will not be further discussed here. As we will see in the following Chapters different implementations proposed deal with this problem and offer widly scalable solutions for the management of shared caches.

Chapter 6

Cache Partitioning Policy Vol. 2

Every day you may make progress. Every step may be fruitful. Yet there will stretch out before you an ever-lengthening, ever-ascending, ever-improving path. You know you will never get to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the climb.

Winston Churchill

6.1 Algorithm Overview

Successful though the results presented by the previous cache partitioning algorithm may have been the fact that it induces a significant hardware overhead to the system makes its use in large scale systems somewhat difficult. In its place another higher level algorithm for the partitioning of the cache is presented. We must also state the fact that the partitioning scheme presented in the previous chapter offers detailed monitoring of the system holding information for each set of the cache which however comes at an accountable hardware overhead. Based mainly on the paper of Qureshi & al [QuP06] this utility based dynamic partitioning algorithm (*UCP*) brings forth the need of more than granting to each application the desired cache space, providing it with the cache partition that it will actually effectively utilize.

In the partitioning policies presented so far, the algorithm tried to appoint to each processor a partition proportional to the extent of the accesses in the cache irrelevantly to whether these data are reused or not. However, the benefit from the use of the cache does not necessarily correlate with the demands of an application.

A most obvious counterexample to that are streaming applications which access large parts of data but only use them once˙ thus not exploiting the main advantage of cache memory. To these and other cases a more important metric to define the size of the partition to be appointed to each core is the extent of use that the thread will actually achieve from the given partition. Therefore the extent of the reuse of already accessed and cached data is crucial in the process of deciding how to share the cache among the processors. We present here a comparison to the LRU policy by addressing the main problem that it issues. In order to do that, we use the miss rates presented when two – different in computing demands – benchmarks execute in the same CMP system sharing the L2 cache. For the purpose of this demonstration we use as before *OLTP (oracle_16cpu_16cl)* and *DSS (db2v8_tpch_qry17_cpu16).* The OLTP workload, presenting a high rate of conflict misses (high reuse rate) has a decreasing number of misses as the ways appointed to it by the LRU replacement policy increase from 1 to 32. On the other hand, the DSS workload presents a lower miss rate as the cache ways change slightly from 1 to 3, however that is pretty much all the benefit it can get from increasing the associativity of the cache since the reuse rate of the cache blocks is bounded. Thus, appointing more cache ways to this application would not benefit its performance. LRU, not differentiating demand from utilization would give a larger number of cache ways to DSS (resulting in a more balanced scheme between the two competing applications) something that would not be accompanied by an increase in performance. As we will see, if UCP was used instead, the miss rate would be significantly lower, and the performance gain would be important.

In order for the system to be able to judge the extent to which an application utilizes its resources it must monitor the access pattern of data as well as the number of misses of each process. The circuitry to do that, must not be power hungry and

must not issue a heavy hardware overhead to the system. Used here, is a novel way to monitor the access pattern of the applications, one that relies on *UMONs* (Utility MONitors) that reassures not to burden the system with a large hardware overhead or a high energy consumption. This monitoring scheme is implemented as follows.

A detailed monitoring of the utilization of the cache would require all possible number of ways per thread to be examined and the corresponding misses to be tracked down. However, this straight-forward approach where each application has as many tag directories as the ways in the cache, each one with the same number of sets as the cache issues a hardware overhead that can not be accepted. This is however not necessary. LRU replacement policy follows the stack property, meaning that is a memory access results in a hit in a cache having N ways it will result in a hit had the cahe had more than N ways. This is an information that we take advantage of when designing the UMONs. Therefore, with a single directory containing as many ways as tha cache associativity (here 32) we can represent the utilization of the cache having as many as 32 ways from a core. Each core is appointed to a certain UMON (Utility Monitor) to monitor the extent of utilization it has in the cache, had the whole ways were appointed to it. Since each application is binded to a core, there are no problems related to applications interleaving with each other, thus causing problems/faulty assumptions in the UMONs' data.

The UMON, composed by a tag array with lines the size of the cache ways has a total size of (cache ways * sets number) and aims to the tracking of the processor's cache utilization had there not been any other competing application in the system. Each one of these entries in the tag array has 32 counters each one representing the hits in this recency position. By increasing the counter that corresponds to the recency position that initiated a hit in the cache at any memory transaction we essentially form the frequency of hits/misses in each one of the recency places in the cache, and

thereof determining the optimal (or close to) number of ways for each application. The remaining transactions that did not result in a hit represent the misses (which are obviously not conflict misses) that would occur in the cache even if all the ways were appointed to this core. By adding to these misses the hits of all the recency positions up to the one we are examining (x) we can calculate the misses that appear if the core has been appointed x ways. This hit/miss distribution formates the utilization function of the core. As we said before the hits tracked in a specific recency position are the misses that would be avoided had the core been appointed one extra way. You cannot know however how this increase will hurt the other applications which is why we use the weighted IPC as a basic metric for the performance increase. Maintaining information on all the recency positions for each set and every core brings forth the issue of the hardware overhead issued in the system by such a policy. Indeed, both the tag directories and the counters used burden the system with additional hardware that can become very expensive and power-hungry. To constraint these defects we issue an alternative approach where the monitoring does not take place in a per-set but in a "per-cache" level.

What we mean is that only one counter per recency position is maintained and the statistics come from the adding of all the hits in that recency position. This modification does not decrease the required tag entries which remain per set but lowers the number of counters from (theSets*theAssociativity) to theAssociativity, which is a trememdous gain for multiple parameters of the system. To make this simplification, it means of course that we accept a decrease in the accuracy of information tracked by the counters but this trade-off as will be seen in the results is not too heavy to bear. A middle solution between *UCP-local* and *UCP-global* relies on statistical sampling (*Dynamic Set Sampling*) of a number of sets (typically 40) which abstain from each other 32 sets. This way, we maintain a representative image

of the utilization of the cache per core while not burdening the system with a huge hardware overhead. In particular the hardware overhead issued comprises a 0.16% of the total size of the cache, a bearable overhead given the improvement in performance. In the present work the two extreme cases (*UCP-local* and *UCP-global*) but an attempt to implement the middle solution is not very difficult as well.

Once we have the statistics for all recency positions for all cores we need to calculate the combined utility taking into consideration the utility functions from all competing cores. To do so exhaustively, is a computing intensive procedure, therefore alternative approaches have been searched. The initial approach was to examine all combinations of way allocation between competing cores to maximize the following function:

$U_{tot} = \sum_{i=0}^{i=num\_cores} U_i(associativity)$, where associativity is the number of ways allocated to each core and Sum(associativity) = ways in cache (here 32).

However, as it becomes clear after a small only increase in the number of cores in the system this algorithm becomes too intensive to follow. Two approaches are proposed to replace the exhaustive search for the optimal allocation of ways.
Both these solutions guarantee an efficient, though not necessarily optimal solution and withhold the computational overhead within reasonable ranges.

The first of these approaches, the *greedy algorithm*, follows the pattern of appointing each way to the application that has the highest utility for that way among all of the existing threads. The comparison of utility functions comes from the results in the counters, whether they come as per set or in a global level. Efficient though it may be, at times the greedy algorithm can present pathological response especially when applications present utility functions that are non-corvex, because though it should the algorithm will stop appointing more ways to them. An improvement to that

would be not to consider the variation of utility between immediate ways but considering more expanded information on the utility function. This is what is attempted by the second algorithm.

That is the *Lookahead Algorithm*, which issues for the first time the concept of *Marginal Utility (MU)*, i.e. the difference in misses when an extra way is appointed to a core. Marginal Utility is defined as:

$$\boxed{MU_a^b = (miss_a - miss_b)/(b-a)}$$

What the algorithm essentially does is find the application that achieves the maximum decrease in misses at the minimum cost of ways and appoint the requested ways to that application. It follows like this until no more applications exist. Another, perhaps more correct at times approach would be instead of minimum number of ways to search for minimum change in ways (b-a) in order to allow applications that need more ways to execute efficiently but by maintaining checks so that no applications will result being starved.

The complexity of the Lookahead Algorithm is $\sim N^2/2$, a great improvement to the previously exponential complexity of the exhaustive algorithm.

In the following experiments we examine two cases: the UCP-local monitoring with the exhaustive algorithm and its opposite UCP-global and the lookahead algorithm. Like this we can evaluate the difference that these simplifications bring to the simulation results. The repartitioning period used is 5 mil cc and these counters (UMONs) are halfed after every iteration to maintain a past information about the utilization of the cache, although for specific applications this number (repartitioning period) can be adjusted. Furthermore, the number of cycles is not the only way to calculate the periods among two consecutive iterations of the partitioning algorithm. Other metrics can be used, with misses being one of the most widely accepted among them. Especially here, where OLTP and DSS workloads are

used, the exploit of miss rates as a frequency for the partitioning algorithm can be a good idea since the number of misses in streams tops every 10,000 misses and a repartition of the shared cache at that moment could help resolve this problem.

6.2 Impementation in Simflex

Implementing this algorithm in Flexus presented the same steps explained in the previous chapter for the Bloom Filter approach. The design is divided in the monitoring scheme and the partitioning algorithm. As we said we implement the UCP-global and UCP-local versions of the algorithm. Their implementation are described in detail in the following sections.

6.2.1 The coreID

As before a coreID field is necessary to determine which core initiated the memory transaction and to update the appropriate UMON. The modifications in the code for its implementation are the same as before, and common among the two tested schemes (global & local).

6.2.2 The UCP-global scheme

6.2.2.1 The Monitoring Scheme

First of all we implement the monitoring scheme based on the UMONs. To do so, we add a new set in the NewCacheArray.cpp corresponding to the new

replacement policy and a set in the NewCacheArray.hpp that will contain the description of the new algorithm (REPLACEMENT_UCP).

In order to test both the schemes that we described we implement a UCP global and a UCP local set. As far as the UCP-global is concerned we search for the core whose block we should evict (choosing besed on the current occupied ways vs. the maximum number of ways it can occupy as before) and update appropriately the counter for the occupied ways. The code for that is as follows:

```
int i = theAssociativity-1;
int lru_ucp_globalListTail ( void ) {
        if (current_ucp_global[coreIdX]>assoc[coreIdX]) {
                while ((coreID[i] != coreIdx) && (i>=0))
                        i--;
        return theMRUOrder_UCP_global[i];
        }
        else {
                while ((coreID[i] == coreIdX) && (i>=0))
                        i--;
                current_ucp_global[coreIdX]++;
        return theMRUOrder_UCP_global[i];
        }
}
```

The entries in the UMONs are updated the moment the memory transaction is resolved, meaning the moment we know if a hit of miss occurred. This happens in the LookupBlock set of NewCacheArray.cpp where the requested block is searched in the cache. Since we implement UCP-global we maintain one counter per recency position for the whole cache per core (*Counter[coreIdX][associativity]*) and since the block is searched in the ordered lru list, finding it updates the counter of the corresponding recency position of the current core. In case of a miss both the cache and the corresponding UMON need to be searched since a miss in the cache does not automatically guarantee a miss in the UMON as well, where the block might still exist. Therefore, for all memory transactions we perform this search and update either the Counter[][] or insert the block in the UMON. Some of the main points of the code described here are the following:

```
LookupResult Set::lookupBlock ( const Tag          aTag,
                                const MemoryAddress aBlockAddress )
 {
  int
    i,
    t = -1;
    int hits;
    int Counter[systemwidth()][theAssociativity];

    for ( i = 0; i < theAssociativity; i++ ) {
          if ( theBlocks[i].tag() == aTag) {
          if ( theBlocks[i].valid() ) {
          hits++;

       return LookupResult ( this, &(theBlocks[i]), aBlockAddress, true );
      }
      t = i;
    }

    if (coreIdX==0) {
      for (j=0;j<theAssociativity;j++) {
            while (Umon_A[theSet][j]!=aTag);
            Counter[coreIdX][j]++;
      }
    }
   else if (coreIdX==1) {
      for (j=0;j<theAssociativity;j++) {
            while (Umon_B[theSet][j]!=aTag);
            Counter[coreIdX][j]++;
      }
   }
   else if …

   if(t >= 0) {
    return LookupResult ( this, &(theBlocks[t]), aBlockAddress, false );
   }
   // Miss on this set
   int misses;
   misses++;
   j=theAssociativity-1;
   if (coreIdX==0) {
      while ((Umon_A[theSet][j]!=aTag) && (j>=0)) {
            j--;
            if (j==-1) {
                  for (i=0;i<theAssociativity-1;i++) {
                        Umon_A[theSet][i+1] = Umon_A[theSet][i];
                  }
            Umon_A[theSet][0]=aTag;
            }
            else {
            Counter[coreIdX][j]++; }
      }
  }
```

6.2.2.2 The Partitioning Algorithm

As far as the actual partitioning algorithm is concerned, as we noted before we first implement the Lookahead algorithm evaluating the marginal Utility based on the Utility functions per core for all possible associativities. First of all, to compute the Utility functions (U[coreIdX][the Associativity]) by appointing them the values f the corresponding counters for each recency position.

To compute the marginal utility we need the number of misses for different associativities per core, however in the current structure of UMONs that corresponds to the number of accesses decreased by the number of total hits (in all recency positions greater than the number of ways appoijted to the core).

$Misses(associativity) = accesses - \sum_{i=0}^{i=associativity} hits(i) = misses + \sum_{i=associativity}^{i=\max\_associativity} hits(i)$. The misses that occur even when the application is given all available ways are *misses* and are a constant for each core (UMON). Either approach in calculating the misses for any given associativity results in a sum of hits for the appropriate recency positions.

As a way to decide on the optimal (or close to) solution for the ways distribution we calculate the max marginal utility for a minimum change in the ways allocated to each processor. To avoid thread starvation additional requirements can be added to the decision algorithm. This exceeds the concerns of the present work.

After the completion of each iteration the Counters' values are halfed to maintain some information about the past cache utilization. This, makes the algorithm more aware of the evolution of the cache accesses, thus better adjusted to the needs of the current thread. Below there are some of the main points in the code that we just descibed, concerning the partitioning algorithm:

```
 {//calculate the sum Utility functions for each core...
for (k=0; k < systemwidth(); k++) {
        for (j=0; j < theAssociativity; j++) {
                for (i=j; i < theAssociativity; i++) {
                        U[k][j] += Counter[k][i];
        }
}

// calculate Marginal Utility function for each associativity LOOK-AHEAD ALGORITHM
    for (i=1;i<theAssociativity;i++)
          for (j=0;j<theAssociativity;j++) {
                MU_0[i][j] = (U[0][i] - U[0][j])/(j-i);
                MU_1[i][j] = (U[1][i] - U[1][j])/(j-i);
                for all cores…
          }

    MU_0_max = MU_0[1][2];
    MU_1_max = MU_1[1][2];
        …

 // find optimal MU for each core..
 while ((sum(assoc[k])) > theAssociativity) {
    for (i=0;i<theAssociativity;i++)
          for (j=0;j<theAssociativity;j++)
                if ((MU_0_max<MU_0[i][j])&& ((j-i)<3)) {
                      MU_0_max = MU_0[i][j];
                      assoc[0] = j; }

    for (i=0;i<theAssociativity;i++)
          for (j=0;j<theAssociativity;j++)
                if ((MU_1_max<MU_1[i][j])&& ((j-i)<3)) {
                      MU_1_max = MU_1[i][j];
                      assoc[1] = j; }
    }
```

6.2.3 The UCP-local scheme

The implementation process is quite similar for the local scheme, therefore only the main differences between the two algorithms will be discussed.

6.2.3.1 The Monitoring Scheme

The only difference between the two monitoring schemes is that now Counters are maintained per set (and every recency position) per core while before there was one Counter per recency position per core. This adds to the hardware overhead, but improves the monitoring capabilities of the system.

Apart from the update of the appropriate counter at a cache hit in the UMON there is no other difference from the previously presented scheme, therefore we will not repeat it here.

6.2.3.2 The Partitioning Algorithm

As far as the partitioning algorithm is concerned we mentioned above that we use here the exhaustive search for the optimal distribution of cores, inducing an exponentially growing computational overhead in the system. This algorithm is clearly not practical in actual systems and is only used here to show the deviation of the results when using less computing intensive algorithms. In order to calculate the total Utility function we need to try all possible combinations of ways' distribution.
The trivial case, in a 2-processor system is running all combinations with one core having i and the other 32-i ways.

To do so, for a multiprocessor system is a much more complicated process. For every i all possible distributions of the remaining cores among the remaining processors must be tested. To do that, we used a recursive function that evaluates all the combinations given the Utility functions of the remaining cores and the number of the remaining ways to be assigned. The starting state of the function is when only two cores have remained which is the situation that we previously described. Every back

step of this function adds one more core to the system, until all combinations are evaluated. After calculating the combined Utility function by a simple max algorithm we find the ways' distribution that achieves the maximum value for it and assign the ways to processors respectively. The repartitioning period is 5 mil cc but it's prone to change. The code for the partitioning algorithm (its main steps) can be found here:

```
//calculate the Utility function per core per set per associativity
for all cores do
        for all sets do
                for all possible associativities do
                        for (i=0; i < associativity; i++)
                        Utility_percore[set][associativity] += Counter_percore[set][i];
        }
  }

  // calculate Total Utility function for each associativity and each set Exhaustive Algorithm
     for all sets do
          for all associativities do
                Utot[set][associativity]  =  Utility_core1[set][associativity]  +
combine(number_of_remaining_cores, number_of_remaining_ways)
// apply recursion to evaluate all combinations for the ways' distribution
}

   // find optimal Utot for each ..
     for all sets do
        for all associativities do
                if Utot[set][associativity] > Utot_max[set] do
                        Utot_max[set] = Utot[set][associativity];
```

As before the Counters' values are halfed after every iteration to maintain past information. All other attributes of the scheme remain the same.

6.3 Results

The experiments are performed for a 16-core system with a 4MB and 32 way L2 cache. The first results are for the UCP-global scheme using the Lookahead algorithm for the partitioning of the shared cache.

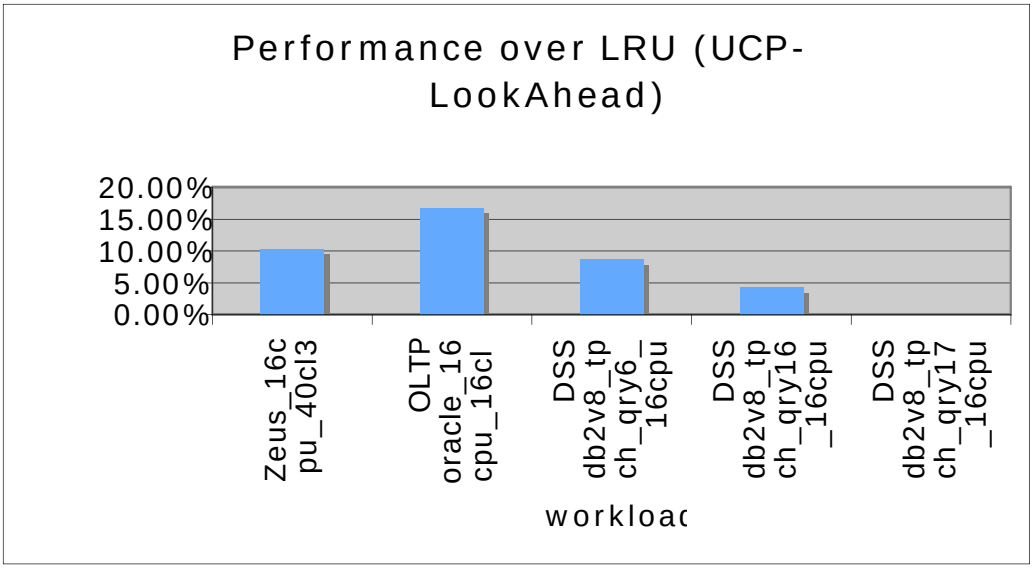

Figure 6.1: Performance over LRU for a 16-core system with a 4M-32way L2 cache (UCP-global, LookAhead Algorithm).

As we can see from the graph above, the system's performance improves 9.00% on average, with the OLTP workload presenting the highest speedup (16.74%). This performance improvement is more intense than in the previously tested ABFCP system, a result coming from the fact that UMONs monitor access patterns more effectively that Bloom Filters. This does not mean that ABFCP cannot outperform UCP, which actually happens for some of the workloads. This deviation is depends on the specific characteristics of each benchmark.

As before, the Web Server and more intensively the OLTP workloads benefit from the cache partitioning algorithm due to their high reuse and sharing rates, while DSS benefit but to a much lower grade. However, we do not see any of the workloads performing worse than when LRU was used, which is a reliable comment that promotes the adoption of this system. The fact that DSS workloads do not present a high improvement in performance derives from their low reuse rate, which result in

them wanting many of the cache's ways to benefit from it, a perspective forbidden by the "starving-prevent" conditions posed in the Lookahead Algorithm.

The following results derive from the second (UCP-local and detailed search) implementation of the UCP algorithm for the same benchmarks as before:

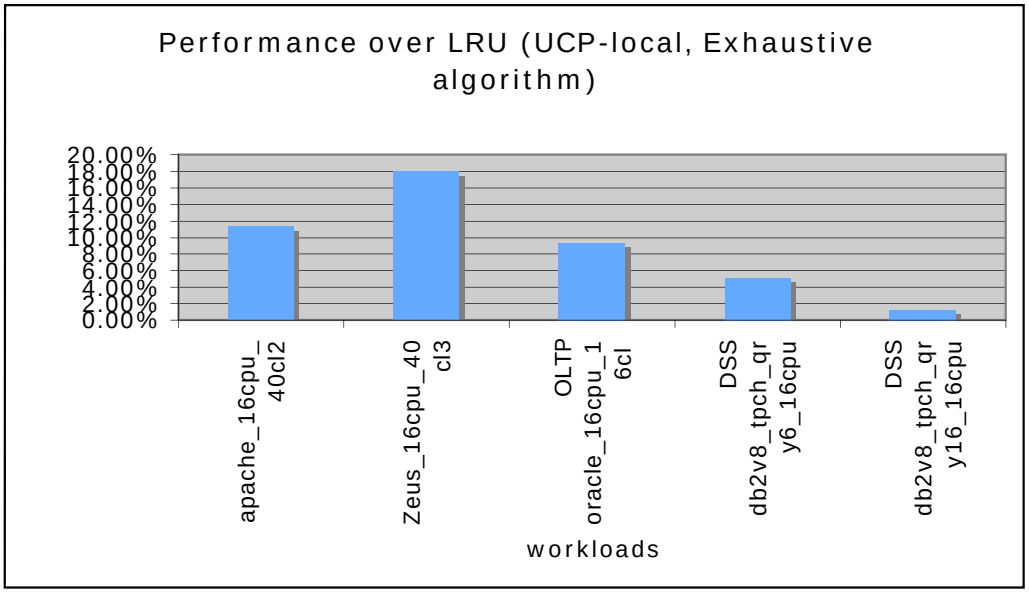

Figure 6.2: Performance over LRU in a 16-core system with a 4M-32way L2 cache (UCP-local, Exhaustive Algorithm).

The difference in the results is clear, but even so, the additional hardware overhead issued in the second case does not justify the adoption of this scheme. The middle solution of sampling appears more appealing to avoid large hardware and computational overhead. We must also note here that this difference in performance (of approximately 1%) mainly comes from the different algorithms used rather than the sampling set of the memory. This, derives from the fact that by only modifying the sampled sets' number the variation in performance speedup is marginal, thus a small number of sets (typically 32) offers approximately the same improvement in

performance as the UCP-global scheme. This result is very important when choosing which of the different implementations of UCP we will adopt.

As we have noticed before, OLTP and DSS workloads - and Web Server to a less extent – present high miss streams rates every 10,000 misses. This logically points to exploiting this feature by assigning the miss rate as a frequency for the partition. Like this, the system (applications) can benefit from the repartitioning of the cache, when this is most needed, i.e. when many concecutive misses appear, misses (conflict mostly) that could have been prevented had the applications that present them have been appointed more ways in the shared L2 cache.

The results from this alternative implementation of the partitioning algorithm appear in the following graph : *(only the global and LookAhead Algorithm have been tested, similar results are expected for the other implementations as well).*

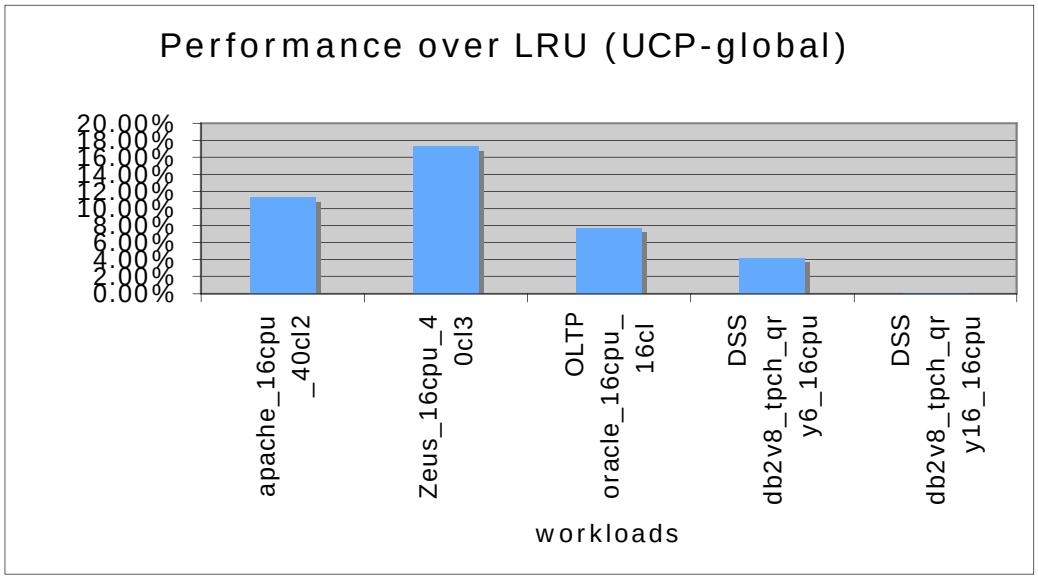

Figure 6.3: Performance over LRU in a 16-core system with a 4M-32way L2 cache (using the miss rate as the repartitioning frequency).

Indeed the results are somewhat improved with an average speedup of 10%, which leads to the consideration of miss bound as a good choice for a partitioning period determinant factor. Generally, there are other changes that could be applied to the present scheme to make it more customized based on the needs – features of the executing applications, but a detailed study of these changes exceeds the purpose of the present section.

6.4 Advantages / Disadvantages

It is clear that the change in perspective in the way we partition the cache, resulted in improved performance, thus making UCP a reasonable choice for systems running multiple simultaneous applications. The hardware overhead as discussed above is existing but not unbearable therefore encompassing this scheme to a computer system (CMP) would not issue serious contradictions between the cost and efficiency of the additional circuitry. Moreover, the fact that UMONs are based on a statistical abstraction rather than detailed (per-set) monitoring of the cache's sets results in this overhead being even less cumbersome for the CMP system, especially when compared to the overhead issued by the previous partitioning policy (Bloom Filter). However, this policy, although improved in relation to previous attempts has some disadvantages mainly related to the level of scalability it can offer when interest shifts from CMPs to larger scale systems.

Then, it is the need for UMONs per core that makes the scheme slightly unfit for many core systems, especially when the UCP implementation chosen is that of the detailed per set monitoring. In those cases, it would be perhaps more wise to either group the monitoring of accesses among cores (specific features of programs

execution patterns would be necessary) or move to a more high-level and thus less hardware intensive policy. Such a scheme is described in the following Chapter and deviates slightly from the concept of cache partitioning.

Finishing here the main cache partitioning policies examined we could make a small note on their efficiency. Both the schemes examined improve the performance of the overall system, to different however extents. And even though they both come with certain requirements from the point of hardware and computational (and potentially power) overhead, their significance is clear when reflecting upon the benefits in cache utilization, thus allowing applications to perform more efficiently in highly parallel and multiprocessor systems.

6.5 Is this enough?

The results presented in the previous subchapter are quite promissing in terms of hardware overhead needed to produce the given speedup. The available cache space is effectively partitioned among the competing applications and the speedup of the overall system justifies the required hardware additions. But the initial *question* remains: Is this enough, and if not is there something better that we can do? In the following section we attempt to give a satisfying answer to this question.

Chapter 7

Cache Partitioning Policy Vol.3

"Reality is a question of perspective; the further you get from the past, the more concrete and plausible it seems -but as you approach the present, it inevitably seems incredible."

Salman Rushdie

7.1 Overview of the algorithm

Presented in this Chapter is a different approach concerning the management of the shared cache space. This scheme instead of partitioning the cache to decide dynamically on the evicted block based on the current partitions concentrates on the way that the blocks are inserted in the cache at a cache hit/miss. The fact that the cache is not actually partitioned i.e. no ways are strictly appointed per core deviates from the standard policies that were presented in the previous two Chapters.

Recapping from the overview of the cache management problem we remind the reader that cache partitioning and adaptive insertion are the main two approaches followed when a shared cache is managed. We are always refering to "virtual" partitioning of the cache, thus there should not be any confusions related to NUCA or relevant schemes. Here we shall present the overview of the proposed algorithm by Jaleel et al. [JHQ+08], while in the following sections the implementation of the algorithm in Simflex and the results collected by a series of experiments will be presented. Finally, in the next Chapter a combination of a cache partitioning algorithm and the adaptive insertion along with the corresponding results is proposed.

As was the case with the cache partitioning policies, adaptive insertion derives from the need that presents as more exigent in modern systems to exploit in the best possible way the available cache space which is shared among competing applications. It does not however follow the already known approach of partitioning the cache to avoid large hardware overheads. Instead, it focuses on the decision making for the placing of the blocks that are inserted in the cache rather than the decision for those evicted from it (although subliminaly in affects this decision as well).

In order to do so, it proposes a certain mechanism that not only controls the insertion of blocks but adjusts to each thread to make the utilization of the available cache space optimal.

We set *four* goals for this work. High-performance, robustness, scalability and low-overhead. In the end, we will come back here to evaluate which of these goals were accomplished and to what level. The goal related to high-performance is quite straight forward since this is the main reason for the initial adoption of the management scheme. Furthermore, since future processors are expected to exceed in number of cores todays' by at least two orders of magnitude, scalability is crucial and should therefore be especially targetted as part of this work. Aside from multiprocessor systems, this cache management – possibly in a higher level of hierarchy – can also be applied to large scale systems whether they concern data centers or supercomputers. Additional to that, a large number of processors implies a complex mix of workloads therefore the proposed scheme must behave acceptably for different types of workloads. Finally, as far as the issued hardware overhead is considered the proposed scheme must bound it within acceptable rates so that the improvement from the cache management does not get compromised by a large storage, power, verification and testing overhead. This work extends the work

proposed by Qureshi et al. [QJP+07]. But with a substantial modification. Recently, a study has shown that adaptive insertion can significantly improve the performance for private caches with marginal hardware overhead. This proposed Dynamic Insertion Policy (DIP) decides between two main insertion policies: the Bimodal Insertion Policy (BIP) and the traditional LRU policy. Each one of these policies has been proven more efficient for certain access patterns but the issue of deciding among them remains. BIP is targetted for trashing applications and treats this problem by inserting a small number of blocks in the MRU position, while inserting the majority of them in the LRU position. Through this method blocks which thrash in the cache will not replace useful blocks (MRU position) but blocks that reside in the LRU (or near) recency position in the cache. In order to decide between these two insertion policies DIP uses Set Dueling Monitors (DSMs). These are used to monitor the behavior of a small fraction of the cache (typically 32 sets) where one of the two insertion policies is exclusively used.

Based on results from this sampling, one of the two insertion policies is chosen. We must note here, that this decision is uniform, which means that one policy will be used in the entire cache opposing to a per set decision making which would be probably more accurate but more computationally intensive. And although this policy is sufficiently accurate for private caches, the same does not apply for shared caches in a multiprocessor system, where each processor (executing application) has different access patterns and block reuse rates. Therefore, a modification to the proposed scheme was necessary. TADIP, short for Thread Aware Adaptive Insertion Policy decides on a per core basis the best of two insertion policies for a shared cache. Again the decision is made for the entire cache blocks but varies between competing cores in the system.

In this implementation the algorithm can actually benefit from the specific characteristics of executing applications, differentiating its response between those with high benefit from the cache and those whose performance is not expected to change by assigning them more privileges in the cache. This is acheved through a monitoring process at a sampled set of cache blocks. To further improve this scheme TADIP-F uses the optimal decision on adaptive insertion policies per core for the executing applications to decide on the insertion policy of any newly executing application. This scheme has been proven to offer a speedup of approximately 17% for a 16-core system like the one simulated here. This is a clear improvement over the previous best policy (UCP) and also over the ABFCP presented in the previous Chapters. To understand the reason behind these results we present here the main steps of the TADIP algorithm.

Aside from the need to better utilize the shared LLC, the motivation behind TADIP is the effort to minimize the hardware overhead inserted in the system by other cache management policies and the urge to exploit the different features applications can present as far as their memory accesses are concerned. Applications can be categorized into 4 families based on their access pattern:

- *Cache Friendly* Applications: Benefit from the cache space and improve their behavior when larger cache space is appointed to them.
- *Cache Fitting* Applications: They need a large part of the cache to perform efficiently, which may at times cause thrashing, especially when they share the LLC with other applications. LRU is usually a good choise for these types of workloads.
- *Cache Thrashing* Applications: These applications cause thrashing effects in the cache, due to the fact that their access space is larger than the cache can

accomodate. And though LRU intensifies the thrashing effect, other replacement policies might be proven more efficient, BIP for example is known to achieve a better utilization of the shared cache when thrashing applications are present.

- *Streaming* Appications: This is the last type of applications that will interest us, since many of the modern applications (e.g. video) belong in this category. These applications do not benefit from the cache, mainly because their reuse rate is non-existent. They access the data that they need only once, thus not benefiting from the existence of the cache. Appointing large cache parts to them is inefficient since it not only doesn't benefit them but results in starving effects for the rest of the executing applications.

The main goal is for the system to benefit from the available cache space no matter what type of applications is executing at any given moment. The traditional way to do this is by assigning small cache fractions to applications that do not benefit from it (such as streaming) and larger fractions to those that do. However, as proven from the information above, to do so more efficiently different insertion policies must be adopted.

Recently, a new method has been proposed to prevent thrashing effects in shared caches. The work conducted by Qureshi et al. [QJP07], proposes that changing the insertion policy from the traditional LRU to BIP results in retaining a fraction of the working set in the cache, thus reducing the conflict misses from thrashing applications. On top of that, BIP results in reducing cache contention, i.e. restricting the cache resources allocated to one application by shortening the lifetime of the lines inserted in the LRU position. Like this, applications that have a working set that exceeds the size of the cache can be constrained through the use of Bimodal Insertion

Policy while others can keep their working set (or a fraction thereof it) in the cache. As we have seen for private caches, the first two types of applications (cache friendly and cache fitting) benefit from the use of LRU, while the other two perform better when BIP is used. When we cross to multiprocessors this approach is a bit simplistic. The traditional LRU replacement policy may cause thrashing, while BIP constraints the working set retained in the cache. To effectively use either one of these policies information on the threads' execution patterns must be available.

DIP, a runtime mechanism that chooses among these two insertion policies based on the workload characteristics presented improved performance for the CMP system. DIP chooses between LRU and BIP by assigning a few sets in the cache to always follow a certain policy and track the misses in them through SDMs (Set Dueling Monitors). DIP uses the winning policy for all the remaining sets – not differentiating among the system's cores - adopting a more coarse grained approach. The choice between the two policies is made through the use of a counter (PSEL), which increases everytime LRU misses and decreases everytime BIP misses. The result of PSEL defines which of the two policies performs better for the given workload. A PSEL value close to zero would favor LRU while PSEL values close to the saturating value would promote the choice of BIP. This is the only information of the threads' execution that the algorithm exploits to decide on the insertion policy. Since no other thread-specific feature is used this algorithm is unaware of the different applications' execution patterns.

When trying to efficiently manage a shared cache, this is not enough. *Thread-Aware Adaptive Insertion Policy* uses statistics from the threads' execution to decide – in a N-core system – on the best insertion policies per core. To do so, optimally one must examine all 2^N possible N-bit binary strings (LRU=0, BIP=1) to decide on the

insertion policies per core. Clearly, this is very inefficient after a number of cores. Therefore, less intensive approaches needed to be found.

First of all, in a mixed workload there are applications whose insertion policies can be decided independently based on their known profiles. As we said before cache friendly applications must use LRU while streaming applications BIP. This reduces somehow the search space. However, this simplification is only possible when we know the profile of the executing applications. When such information is not available alternate approaches are adopted.

TADIP-Isolated attempts to find the best solution by independently finding the decision for the insertion policy, ignoring the co-existence of various workloads in the system. To do so, TADIP uses N+1 SDMs to monitor the behavior of the N threads executing in the system. The reason why for N cores we use N+1 SDMs is that the first one uses LRU for all cores, while the others use BIP for one core and LRU for the rest of them. We decided to use only one BIP insertion per bit string since using more than one, degrades their affect, which is to avoid evicting blocks that are useful in the cache. More than one BIP in an CMP system would be in case more than one applications were streaming which is not a very likely case for most systems. The structure of the SDMs for the default case is <0, 0, 0, 0, …, 0> and <1, 0, 0, 0, …, 0> with the "1" shifting right for the other SDMs (bimodal-SDMs). The difference is only in one among N, policy to make the decisions of TADIP easier. TADIP-I decides on the best insertion policy based on a PSEL counter per application. The samplling set per PSEL is the same as before (DIP) and for every sampled set the SDM policies are applied. In case a block hits in the baseline-SDM (<0, 0, 0, 0, …, 0>) meaning that the transaction misses when the core is using LRU, the counters of all cores are incremented while if a block misses using the bimodal-SDM only the Counter of the corresponding core is decremented. The decision on the insertion policy fo rthe

follower sets is made by the MSB of these Counters for each core. If the MSB in PSEL is zero, LRU used otherwise (decrements in PSEL) BIP is chosen. Using PSEL counters per core gives us the opportunity to notice whether an application is causing thrashing or not, since thrashing applications will have PSEL values close to maximum. As we said before applying BIP to these applications prevents the working set of the other applications in the cache (better utilization) and preserves a part of the working set of this application in the cache without wanting cache space. Therefore, TADIP-I though not directly evaluating the features of other applications to make the decision on the insertion policies, takes into account the impact of thrashing applications (through the monitoring of the corresponding Counters) and applies BIP to them.

There has also been an improved scheme *TADIP-Feedback* which evaluates the insertion policies taking into consideration the execution patterns of all executing threads in the system. This scheme presents improved characteristics as far as performance improvement is concerned with marginal additional hardware overhead (a few additional SDMs), but not deviating a lot from the previously extracted results, leading us to believe that in most of the workloads – especially balanced ones – knowing the behavior of all the threads does not significantly add to the system's performance. This scheme has not been tested here.

As before the simulated system is a 16-core CMP with a 4MB 32 way shared L2 cache. The workloads are the same as in the previous Chapters (Web Server, OLTP & DSS multithreaded workloads).

Fairness issues arise on whether the algorithm can allow a workload to adopt an adaptive insertion policy which results in hurting the performance of other workloads. Related work has dealt with this issue, mainly by applying restrictions to

the adaptive insertions chosen per core to avoid thread-starvation effects by monitoring the change in performance per thread after the decision for the insertion methods has been made, but its detailed introspection is not presented here. Finally, we need to note that since neither the update of PSELs or the insertion logic is in the critical path encompassing this algorithm to the CMP system does not affect cache access latency.

The main problem with adaptive insertion even in its thread aware form is that although it monitors the type of memory accesses (differentiating between thrashing and reused blocks), it does not really provide much information on the extent of memory accesses with the exception of streaming applications. In the latter case, the problem is dealt with the use of BIP to constraint the cache space fraction appointed to this application. The same does not however apply for cache friendly applications. Though LRU is the clear choice, the fraction of the cache that should be appointed to each one of these applications is not clear, since their access extent (working set) is not necessarily of similar size. To face this problem a new combined cache and insertion policy in proposed in the following Chapter.

7.2 Implementation using Simflex

This algorithm unlike the previously implemented cache partitioning schemes does not require additional sets to be added in the preexisting code, since no actual partitioning takes place. What we need to do, is instantiate the sampling set of blocks for the various SDMs and at the moment of fetching the blocks (memory transactions replies), check for misses in the corresponding sets. If a miss is detected in the sampling sets the PSEL Counters are updated. The changes in insertion policies here happen at real-time meaning that follower sets consult the MSB of the corresponding

Counter of the core the set belongs to and adjust their insertion policy respectively. Therefore no partitioning period is necessary. Below, we present the main steps of the thread-aware adaptive insertion algorithm that correspond to the update of the Counters (PSEL).

```
if set belongs sampled_set(0) do
        if miss do
                for all cores do
                        PSEL[core]++;

else
        for all cores do
                if set belongs sampled_set(core) do
                        if miss do
                                PSEL[core]--;
```

Now at a cache miss in one of the follower sets (every set that doesn't belong to the sampled set) we consult the MSB of the PSEL that corresponds to the core that initiated the memory transaction and is it 0 we choose LRU while if it is 1 we choose BIP. The code for that is as follows:

```
if miss do
        if ((PSEL[coreID[set]] && 0xA000)==0)
        // insert at MRU place (LRU replacement policy)
                set = MoveToHead(list)
        else if ((PSEL[coreID[set]] && 0xA000)==1)
        // insert at LRU place (BIP replacement policy)
                set = MoveToTail(list)
```

7.3 Results

The results for the simulated system of 16 cores and 4MB and 32way L2 cache are the following:

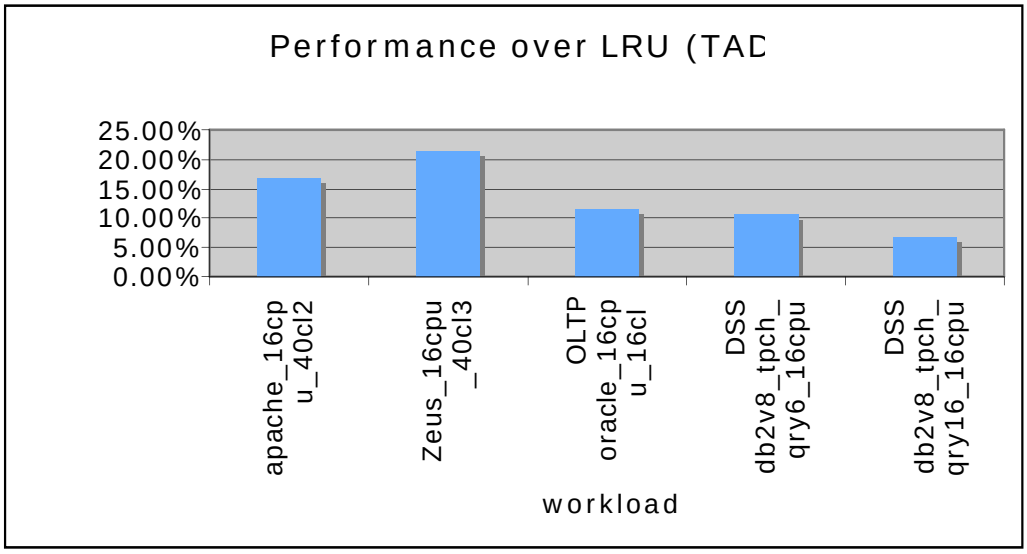

Figure 7.1: Performance over LRU for a 16-core system with a 4MB-32way L2 cache (TADIP).

As we can see from the previous graph TADIP improves the performacne of the simulated system significantly in relation to previously examined schemes. This comes as a result of the fact that the monitoring information is applied in real time, thus adjusting much more accurately to the current features of the executing program. Furthermore, more than before adaptive insertion treats threashing applications by modifying the insertion policy to avoid high miss rates.

The workloads that benefit the most from the use of the adaptive insertion scheme are once more the Web Server and OLTP workloads since they present high reuse rates which are now “protected” by the variation in insertion policy. As for the DSS workloads those too perform more efficiently than before due to the better utilization in the cache, since the use of BIP allows them to maintain a fraction of their working set in the cache, without harming other threads of the workload. The previous schemes did not offer such an improvement in performance due to the fact that the poor utilization these workloads present in the cache led to appointing them very few ways thus not permitting them to run efficiently. The use of TADIP deals

with this problem. Specifically, the use of TADIP-I and TADIP-F resembles thte increase in cache size by 100%, which given the strict constraints for space and power consumption in modern CMPs is a huge advantage of the proposed scheme.

Therefore, it becomes clear that for applications characterized by conflict misses adaptive insertion is a significantly more high-performance algorithm for shared cache management and as such is proposed to be adopted, by systems executing these type of applications and aiming to low overhead and low power consumption solutions for their memory handling.

There are plenty modifications one can propose in the existing scheme. One of the ones that would actually be worth exploring would be the expansion of the search space by considering other SDMs apart from the baseline and bimodal we saw in the previous implementation. This, adds to the computational and hardware overhead of the system but better adjusts to the behavior of the executing programs. The choice on whether to choose such a varioation is at the discretion of the computer architect.

7.4 Pros/Cons

All the above results confirm what was predicted in the beginning of this Chapter. That adaptive insertion policies benefit the system more than cache partitioning policies did. This combined with the fact that the hardware overhead issued in the system by TADIP is marginal leads to realizing that such a system would significantly improve the management of the shard cachce with minimum overhead. In the beginning of the Chapter we posed four goals for this scheme. Efficiency, scalability, robustness and low-overhead have all been faced at different steps of the examined algorithm, and to a more or less extensive degree have been satisfied.

Efficiency, is clearly proven through the results which verify the benefit from

the adoption of this scheme. As for robustness, the scheme's response to different types of workloads, though not optimal is far better than what we have seen before since it offers speedup for all the different types of applications we evaluated here, without harming the performance of any one of them, even when the features of the application are not prone to benefit from the cache to a great extent. If that is the case, it improves to some extent their execution and makes sure that none of the other applications' performances will be harmed by this management. Scalability, is one of the most crucial parameters an architect must consider when evaluating a management scheme.

TADIP, in both its implementations offers scalability prospects, as can be seen from the results for a system with a large number of cores (16-core system) and also from the fact that the increase in the number of cores, only marginally affects the induced hardware and computational overhead with the adding of a few extra Counters. Apart from the low overhead in large scale systems, TADIP has appealing features because its functionality does not degrade as we move to larger number of cores, a defect that was quite obvious in previously examined schemes. All these, combined with the low overheads issued in the system make TADIP very tempting for both smaller and larger scale systems. As we will see in Chapter 9 TADIP is also beneficial from the point of the power overhead it issues in the system which is also marginal. All these, lead to the question. Is the search over? Is adaptive insertion a clear choice for the cache management of CMPs? Although the advantages from the use of this scheme are obvious such an answer would be hasty to give. As we will see in the following Chapter, all the previous policies, TADIP included share some disadvantages that make the choice among them quite difficult. A new scheme that deals with some of these issues is also proposed there.

Chapter 8

Evaluation of the Various Cache Partitioning Policies

If the facts don't fit the theory,
change the facts.
Albert Einstein

8.1 Overview of the results

Presented in the previous chapters are four cache partitioning policies implemented to be used in a CMP system with an L2 (or L3) shared cache. Those policies rely on monitoring attributes of the system to extract information on the executing applications and adjust the management of the cache to the current needs. Two main approaches were presented.

Cache partitioning techniques and adaptive insertion policies. Both philosophies share advantages and disadvantages an overview of which will follow in the current chapter. The results from the adoption of these policies through the use of Simflex simulator have been proven consistent with the assumptions proposed by the corresponding papers and furthermore the simulation time has been proven to greatly decrease as a result of the higher level abstraction implemented by the simulator. The decision over one policy of another depends not only on these results but mainly on the system targeted each time by the algorithm. Therefore, as mentioned above a system with a restricted number of cores where the study of microarchitectural details through a hardware mechanism is manageable can implement more easily algorithms based of lower level metrics of the system to define the partition scheme while in large scale systems such a study would easily fall out-of–hand. It is there where we

rely more on higher level approaches to permit us to manage the shared caches between the competing threads.

In all the experiments conducted for the evaluation of the various cache partitioning policies the metric that defined the period after which the memory space is repartitioned is time and most of the times a constant value of clock cycles is maintained for that scope. Other metrics, like miss rates have also been evaluated and the corresponding results have also been presented in the previous chapters. As a prefetch we can comment that a varying time difference although complies better with the needs of the system comes with the expected overhead of actually modifying the period and for that increasing the overhead (power and performance) of the actual resource allocation algorithm.

The policies presented in the previous chapters are evaluated based on their efficiency, their robustness across various workloads, their scalability and the hardware overhead they issue in the system. We start with a small overview of these algorithms.

8.2 Main policies

8.2.1 LRU

The first policy, implemented by default in Simflex is the LRU replacement policy for all threads in a CMP system. This policy, can be proven efficient for some types of applications such as cache friendly and cache fitting applications which benefit from increasing size in the cache and do not have many conflict misses, but can become very inefficient when streaming and saturating applications are executed.

In these cases, LRU causes high miss rates due to the large working set of

these applications and to the low reuse rate they present. In any case, LRU does not exploit any information of the threads' execution, behaving uniformly for all competing applications. This, however, is not always optimal. For most of the cases, other policies to manage the shared cache offer better performance to the overall system. Some of these policies are summarized here.

8.2.2 Static

The second policy tested in a CMP system with a L2 shared cache was a static sharing of the LLC where all threads have equal rights in the cache space and where allocation comes at a demand-based rate. This policy issues no computational overhead to the system, since no computations are performed for the management of the cache, however its efficiency is unreliable because while in workloads of the same characteristics in terms of memory accesses it behaves acceptably, when the programs demonstrate different access patterns its efficiency is severely lowered. We will see that the following policies allow a far better utilization of the available common cache space.

8.2.3 Adaptive Cache Partitioning using a Bloom Filter

This policy presented in Chapter 5, uses Bloom Filter structures to maintain information about the far-misses a core will have when a certain number of ways is appointed to it. To do so, this policy issues a hardware overhead in the system, an overhead that makes it quite difficult to be adopted for larger scale systems. If that is the scope, modifications to the existing scheme must be conducted, or an alternative policy should be prefered. The other reason why ABFCP is not very suitable for

systems with a large number of cores is that the improvement in performance it offers decreases as the number of cores increases. This decrease though not very dramatic, is expected to degrade the benefits from the use of such an algorithm in a system that is composed by a large number of cores. As far as robustness is concerned, ABFCP performs better for workloads presenting high reuse and sharing rates since as every other cache partitioning policy, it improves the utilization of the cache by avoiding high conflict miss rates (maintaining in the cache the working set that is more likely to be reused). This is the case for Web Server and especially OLTP benchmarks. The same does not apply for DSS workloads where LRU outperforms ABFCP, therefore for such benchmarks we should choose among ABFCP and LRU with LRU being more appealing. Finally, we can say that by switching from a per-set to a more coarse grained monitoring of the system, we can decrease the required hardware overhead, thus making this solution more scalable and efficient.

8.2.4 Utility Based Partitioning

Presented in Chapter 6, Utility-Based Cache Partitioning is a way to partition the ways of a shared cache among competing applications in a way that will increase the utilization level of the cache, by appointing more ways to the applications that will benefit the most from them, i.e. applications with high-reuse rates rather than streaming and saturating programs. To do so, it maintains statistics on the hits in every recency position of the cache and uses those statistics to decide on the partitions per core. UCP is a more efficient algorithm in comparison to ABFCP (average speedup of 10%), mostly because unlike that it does not harm any of the applications that it has been tested for in the present work. Both OLTP and DSS workloads present a speedup though not to the same extent. As expected the workloads that present high

reuse (OLTP) will benefit a lot more from the improvement in the utilizations of the cache, since high conflict miss rates will be avoided. At the same time, starvation for threads is avoided by setting constraints on the number of ways appointed per core. This scheme, is more scalable than the previously tested one, since performance does not degrade as the number of cores increases though the fact that UMONs are appointed per core increases the hardware overhead required after the number of cores exceeds a number. To control this boost in hardware overhead we use sampling techniques to monitor some of the sets, while alternative approaches to further reduce the required hardware are present. In general we can note that this scheme presents some advantages over the previously tested policy, although a cursory choice over one of them would not be correct.

8.2.5 Adaptive Insertion Policies for Managing Shared Caches

The final cache management policy presented was TADIP (Thread-Aware Adaptive Insertion Policy). As we have mentioned before it does not refer to a cache partitioning policy in the conventional way but to the deision making on the way the blocks are inserted in the cache based on the characteristics each one's of the competing applications. To do so, the algorithm maintains counters that help it decide among two insertion policies (LRU and BIP) each one of which benefits different types of programs. By monitoring the behavior of these policies to a sampled set of blocks per core the algorithm decides on the insertion policies for the remaining (follower blocks) in the cache. The improvement in performance it offers is very important (around 17%) and the fact that this performance improves as we move to larger scale systems is perhaps one of the most important features on this scheme.

Scalability, on the one hand and the marginal hardware overhead it issues for

the system are the main attributes that make it so appealing to implement. Moreover, we saw in the results in Section 7.3 that all workloads present improvement in their performance, though again not to the same extent, demonstrating the robustness of the proposed scheme. This comes from the fact that no matter the type of application there is an insertion policy that is efficient for it and that will achieve a good utilization level in the shared cache without hurting the rest of the competing appplications.

This algorithm has been proven more efficient than any one of the previous examined schemes, leading to the impression that it is a clear choice to be adopted for cache management in CMPs. With all its advantages thoughthis algorithm does not come without any disadvantages. The most noticeable among them is the fact that although it successfully evaluates the most efficient insertion policy for each application, to achieve a better utilization of the cache, it focuses more on the pattern of accesses per applications rather than the range of data accessed. That means that given two applications both outperforming for LRU there is no way to control what fraction of the set will be appointed to the one applications over the other, an effect which might lead to starvation of specific threads. The same problem does not apply as much for streaming applications where firstly we do not want them to occupy large portions of the cache and secondly through the use of BIP they retain only a fraction of their working set in the cache. This is not starvation, it is efficient utilization of the available cache space. As for the LRU-like applications however the problem remains. To deal with this problem a new cache partitioning – adaptive insertion policy is proposed.

8.3 Is there an obvious choice?

It must have become clear by now that to decide on the policy used to manage a shared cache is not a straight-forward decision. Many parameters must be evaluated with efficiency, scalability, hardware and computational overhead and power consumption being some of the most important among them. As we have seen the first two policies rely on detailed monitoring to divide the ways of the L2 cache among the different processors. The third algorithm constraints this monitoring process to a great extent and despite that achieves higher performance improvement. This, indeed makes it more appealing for adoption in a CMP system, though there are always attributes that are prompt for improvement. The cache partitioning algorithms were the first one that initiated the subject of managing the shared cache by adjusting to the needs of the executing applications, and opened the way for adaptive insertion algorithms to appear. Therefore their study was necessary to have a clear perception of the evolution of this area and the new challenges posed by the current computer systems.

8.4 A Combined Cache Partitioning + Adaptive Insertion Scheme for Managing a Shared Cache.

Cache partitioning policies issue an important hardware overhead to the system and the performance improvement they offer is bounded but they control the extent of cache accesses a thread can have in the shared cache. Adaptive insertion policies on the other hand, better adjust to the type of accesses competing threads have and offer a significant performance boost to the system, they fail however to

control the fraction of the cache applications sharing the same replacement policy (and in particular LRU) will have. A combination of the two to benefit from each others' adbantages appears tempting. In this Section we will see if it is possible and to what extent it improves the performance of the overall system.

What we want is keep the good features of each one of the two approaches and combine them to achieve a better performance improvement for the CMP system. Therefore, we will implement a cache partitioning policy in order to control and bound the extent of accesses a thread can have in the shared cache and then for each one of these partitions apply an adaptive insertion policy to define how the available per core ways will be utilized. By doing so, we prevent starvation effects for threads having low cache utility since some ways will be appointed to them too, and those ways will be optimally utilized through the appropriate (BIP here) insertion policy.

In this way we aim to resolve the main defects the two approaches present. As far as which cache partitioning policy we will use, we chose to implement UCP using DSS (Dynamic Set Sampling) ot avoid large overhead. The reason why UCP is prefered over ABFCP is the fact that it is more compatible with TADIP enhanced by the fact that since we target a large scale system we want the used schemes to offer efficient and scalable solutions. Though not used here, the possibility of implementing the same algorithm using ABFCP is not excluded and may at times be more efficient. After the decision on the appointed partitions per core inside these partitions we apply the adaptive insertion policy. According to the processor that "owns" a partition we follow LRU over BIP or the other way round. This way, both the space and the recency positions of the cache are efficiently utilized. We must note here that the decision on the insertion policies per core are made in a way similar to the one used before, where a sampled set follows a specific N-bit string of insertion policies and

their miss rates determine the insertion policies usde in the rest of the sets in the L2 cache.

The sampled sets for the two methods should be different so that the one monitoring scheme does not affect the other. Furthermore, the sampled sets for TADIP can be excluded from cache partitioning to make their statistics more pristine from the effect of the partitioning policy, while the contrary is not expected to affect seriously the results of the simulation.

For TADIP we use the baseline and bimodal SDMs explained in Chapter 7 to make the search for the suitable insertion policies less computing intensive.

The implementation of this scheme in Simflex does not deviate a lot from the implementation of the simple UCP and TADIP-I schemes presented in the previous Chapters. The only difference is that now the insertion policy instead of being applied to the whole set is applied to the partition that has been appointed to each core. Therefore, in the same set we may have different insertion policies in the various partitions of the cache. The partitions are assigned uniformly throughout the size of the cache (use of UCP-DSS) and the same applies for insertion policies. They are also assigned for all sets based on the core that initiated the memory transaction . Finally, we note here that in order for insertion policies to be applied in the partitions of the cache, we maintain separate lru lists per core per set in place for the lru list that was maintained per set. This adds a bit to the computational overhead of the algorithm but is necessary for the correct implementation of the desired scheme. The hardwaer overhead induced is that of the UCP scheme since the one issued by TADIP is marginal compared to it.

In the following graph we can see the results from the use of the proposed scheme in a 16-core system with a 4MB 32 way L2 cache shared among all cores. The workloads used are the same as before, Web Server, OLTP and DSS workloads.

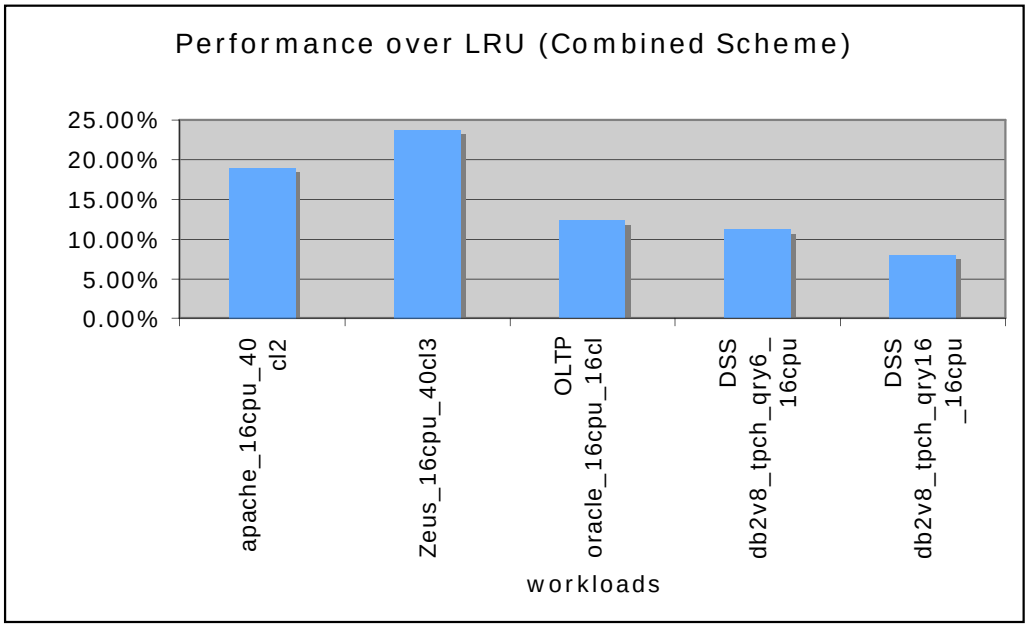

Figure 8.1: Performance over LRU in a 16-core system with a 4MB-32way L2 cache.

We see that the improvement is more intense for Web Server and OLTP workloads where LRU is the most likely insertion policy to be used, while in the DSS workloads the difference is not as intense because of the use of BIP which does not give a lot of cache space to those applications anyway. Therefore, we expect the new scheme to be more efficient for workloads with mixed behavior as far as prefered replacement policies are concerned, i.e. workloads that have threads that benefit from LRU and other from BIP so that the difference from the better utilization of the cache space will be more obvious.

Further improvements of this scheme are possible but a more detailed study of it exceeds the scope of the present work.

An industry example of cache partitioning: The Larabee design

Cache partitioning though an obvious choice for the management of LLCs has not been until recently adopted in modern computer systems. This has changed with the impending release of the Larabee chip by Intel, which supports a scheme for partitioning of the shared cache among the 16 cores of the system. The case here, however, is a NUCA cache tiled among the competing processors while the imbalance of accesses between processors is handled by appointing some space in the partitions of the low-access-rate processors in a demand-based policy.

Larabee is announced to support up to 16 two-threaded in-order cores with vector capabilities connecting with each other though a ring interconnection network while there will also be some specialized graphics hardware embedded in the chip. Each core has a privte instruction and data level 1 cache while the L2 cache is shared among all the cores. Each partition consisting of 256KB keeps a directory partition with information on the core each valid set belongs to, so that a memory transaction can be directed to that core in order to avoid large levels of data replication, though to some extent that is also permissible. It is also announced to support an optimal path search to choose among the cores for additional parts of the partitions to be appointed to processors with higher access rates.

Since at the time of this writing Larabee has not yet been manufactured in silicon the exact details of its design are not known. However, it has become clear that the NUCA cache implemented in it and the parallel rings that connect the various partitions together maintaining a MOESI coherence protocol– though not very scalable for systems with a large number of cores - are very efficient for these purposes.

Chapter 9

The Energy Efficiency Issue

The energy of the mind is
the essence of life.
Aristotle

9.1 Introduction

As systems become more complex and more diverse one problem prevails as a great challenge computer architects must meet up to. The diminishment of the power consumption of modern computer systems, often exceeding 100W for smaller systems and over several MW for today's very large scale systems remains a holy grail, one that needs to be addressed in order to allow the scaling of computer systems. It is well known, that more than the actual components (servers, processors, memories) that constitute a large scale system, their manufacturers are investing on ways to achieve the dissipation and distribution of the power necessary for its function, to prevent the bottleneck in performance that appears otherwise. Moreover, given that the entrance in the multi-core era was accompanied by a step-back in core-level frequency that would allow chip multiprocessors to achieve higher combined throughput at reasonable power, makes the need for no further performance constraints even more exigent. It is more than obvious that such a problem is not to be ignored and must be confronted not only when referring to large scale systems but every computing system in use. Various efforts are made to address this issue, targeting different levels of the system's implementation. Optimizations in processor technology, circuit design, architecture and software are a few of the ways used to control power consumption in

a computing system. The focus here is in the latter either through the OS, the compiler or the user. The main technique relies on adjusting the power state of hw components to match software requirements by putting processors to either idle, nap or sleep mode. A promising fact in this direction is the observation that average and peak energy consumption of a system greatly differ thus allowing the computer architect to benefit from that to decrease the total power consumption of the system. This gap becomes even more clear as the core integration level increases.

Therefore, we consider that a meaningful way to address this issue is by encompassing it in the resource management, mainly focusing up till now on cache partitioning techniques which were presented in the chapters above. It is crucial that the systems knows how to estimate and control its energy consumption and can adjust the tasks executed at any given moment in a way that will permit them to meet these needs with the least possible power consumption. Taking into consideration this gap between the average and peak power consumption one can use techniques to balance the use of different portions (or processors on a die for smaller systems) of the large scale system by assigning them tasks that do not require concurrent peak power consumption (i.e. adjusting to workload fluctuation).

The goal in the present Chapter is not so much the actual management of power consumption in the CMP system but the monitoring of its consumption's fluctuation to verify the effect the previously discussed cache partitioning schemes have on it. The reason for that is that encompassing a cache management policy is important and efficient but to the point where it does not compromise the efficiency of the other computing resources. What we mean is that if a cache partitioning policy results in the increase of the power consumption of the system, its adoption must be seriously reconsidered since the effective management of one resource in the expense of another – especially when that is power – is not acceptable.

Therefore we will evaluate the power consumption is a system that supports the three cache management policies described in the Chapters 5 through 7 and decide on their overall efficiency in the various aspects of the computer system.

9.2 Simulator for Power Consumption

In this chapter a new simulator is introduced. Previously, a separate tool for power consumption studies Flexus-VFI has been encompassed to flexus simulator to permit management of the power consumption in CMP chips. Moreover, it supports OoO execution, an optimization to the previously in order used scheme. A quick overview of the use and functionality of this modules will be presented in the following section. The previously used Simflex simulator does not support power consumption evaluation therefore another road had to be followed. This new version of Simflex runs on top of the preexisting Flexus installation and as such uses the results extracted from Simics. Two new components perform the Power Simulation for the CMP System, through the monitoring of caches, interconnection network and processor state and extracting both maximum and average statistics on power consumption for separate components and the unified system.

9.3 Power overhead from cache partitioning

An important aspect one must take into consideration when encompassing cache partitioning techniques in a computer system is the power consumption overhead that these schemes issue, and when the system is a high-performance server those increases in the energy consumption though not welcomed are bearable, but when the system is a low-power embedded system this overhead is not to be ignored.

As far as the energy consumption is considered one must take into consideration the additional power resulting from the hardware overhead issued in the system, the access rate in this circuitry and whether this is externally controlled (a small controller to implement the partitioning) or affects the critical path.

This is the one aspect of this issue. To consider cache partitioning as a power hungry technique is to forget its results in the utilization of the cache space. Causing the cache to have an improved utilization cache partitioning schemes result in fewer accesses in the lower levels of the memory. This, of course is expected to cause a decrease in the power consumption of the overall system. In order, to confirm or disprove these assumptions we use the PowerTracker components of Simflex to evaluate the energy consumption when various partitioning schemes are used.

Power Consumption in Adaptive Bloom Filter Cache Partitioning

Studying the first of the three implementations for cache partitioning in a CMP system we turn our focus in estimating the power consumption issued in the experimental system by the use of ABFCP. What we attempt here is not to orchestrate a power management – this is the scope of Chapter 10 – but instead to evaluate the increase or decrease in power consumption because of the use of the partitioning scheme in the shared cache. The hardware overhead as explained in Chapter 5 consists of a Bloom Filter for each set and each of them consisting of k bits where k was assigned the values of 5 and 6. Using Bloom filters, however, incurs some additional power for reading and updating the filters for every memory instruction. Using the 130nm ASIC synthesis methodology described in the Alpha evaluation section, the capacitance of the Bloom filter was 64pF which roughly is the capacitance of a 12-entry, 40-bit unfiltered CAM.

This structure resembles in power consumption that of a small cache of a corresponding size and with an access rate the same order as the miss rate in the LLC. The extent of this consumption ranges to mW and although undesirable in a system it does not cause significant problems as far as heat dissipation or energy awarness are concerned, since it corresponds to less than 2% of the total consumption caused by the accesses in the L2 cache. Moreover, the cache partitioning scheme induces a power consumption as a result of the extra bit tags that relate each block in the L2 cache to the processor that issued its access. Overall, considering the lower power consumption from the more infrequent main memory accesses, we deduce that ABFCP slightly increases the power consumption of the system, a feature which is unwanted though not extremely determinant in the choice of the partitioning algorithm. Had this policy have a higher efficiency the increase in power from the BFAs could have been compensated by the decrease in power consumption from the less frequent accesses in main memory.

Power Consumption in Utility-Based Cache Partitioning

We can resemble the UMON's power consumption as that of a small cache the size of [(tag)*(associativity)*(sampled sets)] which according to relative studies is estimated not to exceed a few mW with the access rate of the shared L2 cache.

Furthermore, the fact is that the better utilization in the shared cache results in fewer accesses in the main memory which saves the system from a large power consumption overhead. These savings are more obvious here than for ABFCP due to the higher performance improvement that this scheme offers. This leads in the realization that the cache partitioning scheme based on UCP does not burden the system with additional power consumption at least to the point where that becomes a

problem for its functionality. Therefore, if the additional hardware required is not the limiting factor in the examined system this partitioning scheme is a good choice for the management of the L2 cache.

Power Consumption in Adaptive Insertion Policy

Finally, we evaluate the impact in power consumption of the use of the adaptive insertion policy presented in Chapter 7. As we saw before, cache partitioning policies induce an overhead as far as energy is concerned mainly because of the extra traffic they issue while monitoring and repartitioning the LLC. Here however, the case is quite different. The hardware overhead issued in the system by the use of adaptive insertion is marginally lower than the one issued by either one of the previous cache partitioning policies (ABFCP and UCP). This leads in a much lower power consumption in the system than the one induced before, thus not increasing the overall power consumption of the CMP system.

The accesses in the L2 cache are significantly reduced which leads in a lower power consumption from that point of view. Therefore, cache partitioning if designed carefully and economically can lead not only to a better utilization of the available cache space but moreover to the reduction in the power consumption of the monitored system.

9.4 Assumptions

What can be derived from the results presented in the previous section is that various cache partitioning schemes can have different effect in the power consumption of the CMP system. ABFCP has been proven to increase power consumption while in UCP the increase in power consumption from the use of UMONs is somewhat balanced by the decrease from the lower number of main memory accesses. Finally, in TADIP the power consumption is lowered which is one extra reason that intensifies our belief that adaptive insertion policies, either on their own or combined with a cache partitioning scheme are more suitable for adoption in modern CMP and larger scale systems. In any case, as has been proven here resource management is not a one sided problem and in order to ensure that the proposed scheme improves the functionality of the system without severely hurting some of its parameters there are additional aspects that should be examined, with power consumption being one of the most important of them.

Chapter 10

Future Research Interests

My interest is in the future because I am going to spend the rest of my life there…
Charles Kettering

10.1 Overview

The area of resource management in both small and larger scale computer systems being an ever changing field has many new areas of research to offer. As we are concluding the study performed as part of this diploma thesis there are some subjects that capture our interest and a short presentation of each of them is what will encompass the subject of this chapter. The areas are stated in an order that is believed to be one in which they can be addressed more efficiently and in which they are more likely to give valuable to further research results.

10.2 The Energy Efficiency Problem

Although this issue has already been addressed in the study above we believe it to be one of the most crucial areas of research in the years to come. Achieving a decreased to a large extent power consumption in either embedded or – even more – large scale systems remains an open problem one that relies to an efficient allocation of the available resources to be resolved. This allocation will be effective provided that the system will know its needs according to the tasks executed at any given moment and will therefore not spend unnecessary power by performing all the processes at maximum performance, but rather manage the execution pattern so that

higher priority or computing intensive applications are granted the necessary computing resources (among which is power) to complete while other tasks can be performed at a less power consuming mode.

In this direction several studies have been proposed. Most of them rely on monitoring portions of the system in order to allow it to have a full perspective of the processes executed at any moment and through this power-on or power-off parts of the system that run the corresponding tasks. It is commonly observed that when the system does not perform at maximum power consumption but energy consumption fluctuates in time the final power consumption is much lower compared to the initial even if this comes with an additional time overhead (lowers the overall performance). The extent in which this performance overhead is to be accepted by the system designer depends on the requirements that apply for any given system.

10.3 Allocation of thread–specific processors in large-scale systems

10.3.1 Use of Message-passing Algorithms

The following area related extensively to large scale systems though it may be can also be applied with some simplifications to smaller multiprocessor and multicore systems.

The main purpose of this idea is to consider the actual cores as a resource that can be distributed among the competing threads executing in a computer system. This of course has meaning when the system consists of various types of processors, accelerators, GPUs and vector processors among them.

It is safe to presume that special purpose processors will present with a much higher performance in specific types of applications rather than if these applications

where to be executed in general purpose cores. This relies on the special architectural features (ISA, architecture optimization, data access) that these processors present and which benefit specific families of applications. Of course, special purpose computing comes with the price of general functionality since it can only be applied for these types of processes. However, in a system as the one that interests us here this is not to be considered as a problem since GPUs are available to execute general purpose computing tasks, therefore we can benefit from heterogeneity to improve performance and power management. Related work on the area has been presented by (Tullsen & al.) targeting multiprocessor systems of up to 16 cores and using monitoring techniques for the executing threads to define which of the available cores in the system would be most effective for each one of the competing applications.

This core distribution is believed to be applicable and more effective in large scale systems where the range of the executing applications greatly varies and where different types of cores are usually available. Of course, when large scale systems are considered other issues must be taken into consideration with the one creating the greatest bottleneck being the actual way of communicating the results derived by the traces of each thread to the various cores of the system to find the one that suits them best. Here is that message-passing algorithms already used in large scale systems can be applied to make this distribution manageable. Other than that, the search space for the appropriate core in the system remains prohibitive thus promoting the idea of partitioning the actual system (groups), subdiving it in smaller groups of cores which makes it easier for the algorithm to find an acceptable though not always optimal solution. In case, a better solution is required one can always reform the specifications of the system's grouping to allow the algorithm to expand its search space and find a potentially better final solution.

The main reason why such an issue was not further explored in the present work in dual. First of all, the lack of support for different processor types by the Simflex – Simics simulator made it difficult to implement such a scheme effectively.

Furthermore, the reasonable overhead that the encompassing of these message-passing algorithms presents is bearable for systems consisting of a large number of processors, but can be quite annoying for smaller systems as those examined here. Therefore, we believe that valuable though the use of such algorithms may be it must be carefully evaluated in order to prevent computational and communication overheads that will hurt rather than improve the performance of the whole system. As stated above, in large scale systems the seeking for an appropriate core to execute a thread can be localized with a global manager resolving issues related to poor thread distribution or highly unsuitable cores within a given group of processors. In such cases, further approaches can be proposed, but that exceeds the scope of the present work.

10.4 Nash Equilibrium

The increasing complexity of computer systems forces computer architects to treat threads as actual players in a multiplayer game in which the optimal performance of the given system under some requirements is the scope of the game. In order, to accomplish this goal threads must request the resources needed to perform their necessary computations. As stated in the corresponding chapter, to do so, the threads must more than request the optimal condition of resources isolated from the rest of the system, develop a behavior that will allow them to lean towards an equilibrium, one where the requests of each thread are optimal for the total system, given the demands of every other process. Provided that neither cooperations between the threads exist,

nor any thread will modify its initial request this Equilibrium is proven to exist for each system of n-players. Now, as presented in the same chapter an optimal combination of requests might not exist for each set of competing threads, or at least not a deterministic one, as expressed above. There is where the differentiation between the pure and mixed (or probabilistic) Nash Equilibrium in multi-player games rises, while looking for the consensus of all the "players" involved in the sharing of the computing resources.

10.5 Interconnection Network + Security detection and resolution

10.5.1 Interconnection Network Management

This next section primarily relates to multiprocessor systems that communicate through a type of an interconnection network, it can however be expanded to systems of a larger scale in which the interconnection network is viewed either partially or in a global manner.

As the resources presented in the chapters above, the interconnection network consists of a valuable commodity that should be shared among the different cores (or threads) of the system in order to achieve the different requirements whether they concern fairness or priorities for each task.

10.5.2 Security Attacks Prevention and Resolution

Though not the typical form of a computing resource, security either from inner or outer threats is a resource not to be ignored especially in the era of modern

computing systems. The opportunity to handle it as part of the interconnection network management rises now more promising than before. And this comes as a result of the fact that communication targeting the sharing of the interconnection network travels troughout the system pinging all cores thus detecting potential attacks that might compromise the system.

In this direction previous work [Set07] has been conducted mostly relating to exploring the potential of a hardware security scheme in which the detection of attacks is encompassed to the interconnection distribution scheme thus reducing the overhead and providing the system with the necessary for its function security.

Though the actual resolution of security attacks detected with the scheme proposed in the publication above exceeds the scope of this work methods exist to make this error resolution possible.

10.6 Scheduling Issues (Time as a resource)

In all of the work conducted up till now, resources consist of memory, bandwidth, rights in the interconnection network, energy and processor types. Still, however, the most important commodity in a computer system remains time, and its allocation pattern to different threads is an issue of great importance that every architect must in one way or another face. Scheduling, is therefore and issue of critical importance and as such must be handled in order to offer ways to improve the performance of a given computer system.

10.7 TM and resource management in multiprocessor systems

Transactional memory has recently presented as a way to efficiently manage widely parallel programs without facing with the difficulties of parallel programming. Doing so, it ensures safe and efficient memory accesses, which could affect or be affected by the partitioning scheme implemented in the shared L2 cache.

10.8 Additional Thoughts.

In all the areas before, we presented research interests focusing on a single resource each time. The challenge however remains: Is there an efficient way to combine the management of various computing resources through a high-level system running on top of a large-scale system thus achieving an improved performance for the overall system? This is believed to be a very crucial question, whose answering could have an important impact in computing systems.

Chapter 11

Concluding Remarks

There is at the back of every artist's
mind, a pattern or a type of architecture.
Gilbert Chesterton

The previous chapters have covered an overview of the problem of resource management either in the form of cache partitioning algorithms or energy consumption constriction between threads running concurrently in a CMP system. This chapter is an overview of the already presented work and a short conclusion of the results depicted from it, as well as some last future research interests on the subject of resource management and especially in the area of large scale systems.

Starting by presenting a short overview on the area of managing the computing resources in the environment of a multicore system we implemented a series of cache partitioning algorithms starting with a static "partitioning" of the LLC and continuing to dynamic schemes (using a Bloom Filter and Utility Based Monitoring) evaluating the performance gains in the system that each of these algorithms offer. Moreover, a different approach on the subject of cache partitioning supported by adaptive insertion was presented. As was made clear by the results presented in the related chapters both these algorithms offer an improved memory management each one of them to a different extent and with a different overhead for their implementation. Therefore, a clear choice between them can not be made easily and the system designer must base his decision on the specific needs of the computer system. Furthermore, we presented a new scheme based on the combination of cache partitioning and adaptive insertion benefiting from the advantages of these two

approaches and seeking to deal with some of the problems that they present. This new scheme, has presented improved results over all the previous implemented policies.

Moreover, as we proceeded in the study of resources management it became clear that managing one resource in detail must not lead in completely ignoring the rest of the computing resources, with power being one of the most important of them.

All related results have been implemented and tested using the Simflex simulator to benefit from the opportunity to decrease the simulation time by 3 – 4 orders of magnitude. Specifically for the power consumption management an extended version of Simflex (Simflex-vfi) was used to offer dynamic and static power modeling of the CMP system.

The workloads used, were multithreaded workloads from the SPEC*Web*99 benchmark suite consisting of Server, OLTP and DSS workloads with different characteristics as far as their memory needs (sharing and reuse) are concerned.

Therefore, to conclude this dissertation it is safe to predict that resource management either in the sense of targetted management of one type of resource or - more - in the sense of combining the utilization of different types of resources will consist of a field with increasing importance in the years to come not merely to improve the performance but the overall functionality and scalability of computer systems.

Bibliography

[ABD+03] D. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook and S. Schuster. Dynamically Tuning Processor Resources with Adaptive Processing, *IEEE Computer Magazine*, 36(12):49-58, December 2003.

[AGP+05] A. Ailamaki, K. Gao, I. Pandis, B. Falsafi, B. Gold, M. Shao, J. Hoe, N. Hardavellas, J. Smolens, J. Kim, S. Somogyi, T. Wenisch, R. Wunderlich. SimFlex: Fast, Accurate, and Flexible Simulation of Computer Systems. *SimFlex Tutorial, MICRO-38*, November 2005.

[AGP06] A. Ailamaki, K. Gao, I. Pandis, B. Falsafi, B. Gold, M. Shao, J. Hoe, N. Hardavellas, J. Smolens, J. Kim, S. Somogyi, T. Wenisch, R. Wunderlich. SimFlex: Fast, Accurate, and Flexible Simulation of Computer Systems. *SimFlex tutorial, ISCA-33*, June 2006.

[Alb98] D.H. Albonesi. The Inherent Energy Efficiency of Complexity-Adaptive Processors. In the *Proceedings of the 1998 Power-Driven Microarchitecture Workshop*, p. 107-112, 1998.

[BA97] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *CS TR 1342*, University of Wisconsin-Madison, June 1997.

[BTM00] D. Brooks, V. Tiwari, M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In the *Proceedings of the 27th International Symposium on Computer Architecture (ISCA00)*. Page(s): 83 – 94, June 2000, Vancouver, British Columbia, Canada.

[CFH08] E. Chung, M. Ferdman, N. Hardavellas. SimFlex and ProtoFlex: Fast, Accurate, and Flexible Simulation of Computer Systems. *SimFlex & ProtoFlex tutorial, PACT08*, 25 October 2008.

[CFG07] E. Chung, M. Ferdman, B. Gold, N. Hardavellas, J. Kim, I. Pandis, M. Shao, J. Smolens, S. Somogyi, E.Vlachos, T. Wenisch, R. Wunderlich, A. Ailamaki, B. Falsafi and J. C. Hoe. Flexus Getting Started Guide, *Carnegie Mellon University Release Notes*. November 2007.

[CML03] A. Cristal, J. F. Martinez, J. Llosa, and M. Valero. A case for resource-conscious out-of-order processors. In *IEEE Computer Architecture Letters*, Vol. 2, Oct. 2003.

[CRV+04] F. J. Cazorla , A. Ramirez , M. Valero , E. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors, In the *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture (Micro04)*, p.171-182, Portland, Oregon, December 04-08, 2004.

[CY06] S. Choi, D. Yeung. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In the *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA06)*. p:239-251, 2006.

[DKK06] M. Dalton, H. Kannan, C. Kozyrakis. Deconstructing Hardware Architectures for Security. *5th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) at ISCA06*, Boston, MA, June 2006.

[DY02] G. K. Dorai , D. Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance, In the *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT02)*, p.30, September 22-25, 2002.

[EMP09] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. *to appear in Proceedings of the 15th International Symposium on High-Performance Computer Architecture (HPCA09)*, Raleigh, NC, February 2009.

[Fal09] B. Falsafi. SimFlex & ProtoFlex: Full-System Emulators/Simulators for Large-Scale Multiprocessors. *PARSA Parallel Systems Architecture Lab, EPFL.*

[GHK+09] B. Grot, J. Hestness, S. Keckler, and O. Mutlu. Express Cube Topologies for On-Chip Interconnects. *to appear in Proceedings of the 15th International Symposium on High-Performance Computer Architecture (HPCA09)*, Raleigh, NC, February 2009.

[GKS+08] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, D. Burger. Multitasking Workload Scheduling on Flexible Core Chip Multiprocessors. *Workshop on Design, Architecture and Simulation of Chip Multi-Processors, PACT08*, December 2008.

[GKS+08] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, D. Burger. Multitasking Workload Scheduling on Flexible-Core Chip Multiprocessors. In the *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT08)*, October 2008.

[HRI+06] 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 *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–22, New York, NY, USA, 2006. ACM.

[IBC+06] C. Isci, A. Buyuktosunoglou, C. Cher, P. Bose and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies:Maximizing Performance for a Given Power Budget. In the *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (Micro06)*, pages 347-358, Orlando, US, December 2006.

[ICM06] C. Isci, G. Contreras and M. Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In the *Proceedings of the 39th ACM/IEE International Symposium on Microarchitecture (MICRO-39)*, December 2006.

[IMM+08] E. Ipek, O. Mutlu, J. F. Martínez and R. Caruana. Self Optimizing Memory Controllers: A Reinforcement Learning Approach. In the *Proceedings of the 35th International Symposium on Computer Architecture (ISCA08)*, pages 39-50, Beijing, China, June 2008.

[IMM+08] E. Ipek, O. Mutlu, J. F. Martinez, R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In the *Proceedings of the 35th International Symposium on Computer Architecture (ISCA08)*, Beijing, China, June 2008.

[Iye04] R. Iyer. CQoS: a framework for enabling QoS in shared caches of cmp platforms. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004. ACM.

[IZG+07] 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. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, New York, NY, USA, 2007.

ACM.

[JHQ+08] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr., J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In the *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT08)*, p:208-219, Toronto, Canada, November 2008.

[JPL08] N. Enright Jerger, L.-S. Peh and M. Lipasti. Circuit-Switched Coherence. *Network on Chip Symposium*, April, 2008.

[KBK02] C. Kim, D. Burger, S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In the *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS02)*, 2002.

[KCS04] S. Kim, D. Chandra and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In the *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT04)*, p:111-122, September 2004.

[KDO08] M. Mercaldi Kim, J. D. Davis and M. Oskin, T. Austin. Polymorphic On-Chip Networks. *International Symposium on Computer Architecture (ISCA-35)*, June 2008.

[KFJ+03] R.Kumar, K. Farkas, N. Jouppi, P. Ranganathan and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In the *Proceedings of the 36th International Symposium of Microarchitecture (Micro03)*, p. 64-76, San Diego, December 2003.

[KHM01] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In the *Proceedings of the 28th Annual International Symposium on Computer Architecture, (ISCA01)*, pages 240–251, 2001.

[Kim08] Martha Kim. Brick and Mortar Chip Fabrication. *Doctoral Dissertation*, 2008.

[KKM05] M. Kırman, N. Kırman, and J. F. Martinez. Cherry-MP: Correctly integrating checkpointed early resource recycling in chip multiprocessors. In *Intl. Symp. on Microarchitecture (Micro05)*, Barcelona, Spain, Dec. 2005.

[KMO07] M. Mercaldi Kim, M. Mehrara, M. Oskin, T. Austin. Architectural Implications of Brick and Mortar Silicon Manufacturing. In the *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA-34)*, June 2007.

[Koz02] Christoforos Kozyrakis. Scalable Vector Media-processors for Embedded Systems. *Dissertation Thesis.* University of California at Berkeley, May 2002.

[KSG+07] C. Kim, S. Sethumadhavan, M.S. Govindan, N. Ranganathan, D. Gulati, D. Burger and S.W. Keckler. Composable Lightweight Processors. In the *Proceedings of the 40th Annual International Symposium on Microarchitecture (Micro 2007)*, Chicago, IL, USA, December 2007.

[LGF01] K. Luo, J. Gummaraju and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, November 2001.

[LIM09] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. *to appear in Proceedings of the 36th International Symposium on Computer Architecture (ISCA09)*, Austin, TX, June 2009.

[LM06] J. Li, J.F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In the *Proceedings of the Intl. Symp. on High-Performance Computer Architecture (HPCA05)*, Austin, TX, Feb. 2006.

[LMD05] J. Li and J. F. Matrinez. Power-performance considerations of parallel computing on chip multiprocessors. In *ACM Trans. on Architecture and Code Optimization*, Vol. 2, No. 4, Dec. 2005.

[LMM05] J. Li and J. F. Martinez. Power-performance implications of thread-level parallelism in chip multiprocessors. In *Intl. Symp. on Performance Analysis of Systems and Software*, Austin, TX, Mar. 2005.

[LMN+08] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-Aware DRAM Controllers. to appear in the *Proceedings of the 41st International Symposium on Microarchitecture (MICRO08)*, Lake Como, Italy, November 2008.

[MA03] A. El-Moursy and D. H. Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In the *Proceedings of the 9th Symposium on High-Performance*

Computer Architecture (HPCA2003). February 8-12, 2003, Anaheim, California, USA.

[MBH02] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal, 6(1)*, February 2002.

[MCC+07] A. McDonald, B. D. Carlstrom, J. Chung, C. C. Minh, H. Chafi, C. Kozyrakis, K. Olukotun. Transactional Memory: The Hardware-Software Interface," *IEEE Micro, Special Issue on Top Picks from Architecture Conferences*, vol. 27, no. 1, January/February 2007.

[MIB08] J. F. Martinez, E. Ipek, R. Bitirgen. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In the *Proceedings of the 38th Annual International Symposium on Microarchitecture (Micro 2008)*, November 2008.

[MSB+05] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[MSP+06] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, S. Eggers. Instruction Scheduling for Tiles Dataflow Architectures. In the *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, October 2006.

[QSP07] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In the *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA07)*, 2007.

[QTP05] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In the *Proceedings of the 32th International Symposium on Computer Architecture (ISCA05)*, 2005.

[MTD99] J. F. Martinez, J. Torrellas, and J. Duato. Improving the performance of bristled CC-NUMA systems using virtual channels and adaptivity. In *Intl. Conf. on Supercomputing*, Rhodes, Greece, June 1999.

[MuM09] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Memory Controllers. *to appear in IEEE Micro, Special Issue: Micro's Top Picks from 2008 Computer Architecture Conferences (MICRO TOP PICKS)*, January/February 2009.

[MuM08] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In the *Proceedings of the 35th International Symposium on Computer Architecture (ISCA08)*, pages 63-74, Beijing, China, June 2008.

[Mut06] Onur Mutlu. Efficient Runahead Execution Processors. *Dissertation Thesis*, University of Texas at Austin. May 2006.

[Nas50] John Nash. Non-cooperative Games. *Dissertation Thesis*. May 1950.

[NHG08] K. Nikas, M. Horsnell, J. Garside. An Adaptive Bloom Filter Cache Partitioning Scheme for Multicore Architectures, In the *Proceedings of the VIII International Symposium on Systems, Architectures, Modeling and Simulation (SAMOS08)*, July 2008, Samos, Greece.

[NLS07] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. *SIGARCH Computer Architecture News*, 35(2):57–68, 2007.

[PH08] David Patterson, John Hennessy. Computer Architecture: Hardware/Software Interface. 4th edition. *The Morgan Kaufmann Series in Computer Architecture and Design*, 2008.

[PH02] David Patterson, John Hennessy. Computer Architecture: A Quantitative Approach, Third Edition. *The Morgan Kaufmann Series in Computer Architecture and Design*, 2002.

[PKK+06] P. Petoumenos, H. Z. G. Keramidas, S. Kaxiras, and E. Hagersten. Statshare: A statistical model for managing cache sharing via decay. In *Second Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2006)*, 2006.

[PKZ+06] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. Modeling cache sharing on chip multiprocessor architectures. *IEEE International Symposium on Workload Characterization 2006*, pages 160–171, Oct. 2006.

[QLM+06] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. *HPS Technical Report, TR-HPS-2006-003*, University of Texas at Austin, February 2006.

[QMP05] M. K. Qureshi, O. Mutlu, Y. N. Patt. Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors. In the *Proceedings of the International Conference on Dependable Systems and Networks (DSN05)*, pages 434-443, Yokohama, Japan, June 2005.

[QSR09] M. K. Qureshi, V. Srinivasan and J. A. Rivers. Scalable High-Performance Main Memory System Using Phase-Change Memory Technology. To appear in the *Proceedings of the 36th International Symposium on Computer Architecture (ISCA),* Austin – Texas, June 2009.

[QuP06] M. K. Qureshi, Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In the *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (Micro06)*, Florida, USA, December 2006.

[RLT06] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–12, New York, NY, USA, 2006. ACM.

[RN02] Stuart Russell, Peter Norvig. Artificial Intelligence : A Modern Approach. 2nd Edition. *Prentice Hall*, December 2002.

[RR03] S. E. Raasch and S. K. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In the *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT03)*, 2003.

[RRP+07] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In the *Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA07)*, Phoenix, AZ, February 2007.

[SBK06] S. Sethumadhavan, D. Burger, and S. W. Keckler. Partition the Banks, not the functionality, of Large-Window Load-Store Queues. *Department of Computer Sciences, The*

University of Texas at Austin, Techical Report TR-06-39, 2006.

[Set07] Simha Sethumadhavan. Scalable Memory Disambiguation. *Dissertation Thesis*. The University of Texas at Austin, December 2007.

[SKI08] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *SIGARCH Computer Architecture News*, 36(1):135–144, 2008.

[SMK+07] S. Srinath, O. Mutlu, H. Kim and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In the *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA07)*, pages 63-74, Phoenix, AZ, February 2007.

[SMQ+09] M. A. Suleman, O. Mutlu, M. K. Qureshi and Y. N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. To Appear in the *Proceedings of the IX International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS09),* 2009.

[SPM+06] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, S. Eggers. Area-Performace Trade-offs in Tiled Dataflow Architectures. In the *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA-33)*, June 2006.

[STV02] A. Snavely, D. M. Tullsen, G. Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. *SIGMET02*, 2002.

[SQP08] M. A. Suleman, M. K. Qureshi, Y. N. Patt. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-threaded Workloads on CMPs. Appears in the *Proceedings of the VIII International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS08),* 2008.

[WCN07] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis and K. Olukotun. A Practical FPGA-based Framework for Novel CMP Research. In the *Proceedings of the 15th ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays (FPGA)*, Montery, CA, February 2007.

[WKK08] H. Wang, I. Koren and C. M. Krishna. An Adaptive Resource Partitioning

Algorithm in SMT Processors. In the *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT08)*, October 2008.

[WM08] C.-J. Wu and M. Martonosi. A Comparison of Capacity Management Schemes for Shared CMP Caches. In the *Proceedings of the 35th International Symposium on Computer Architecture (ISCA08)*, Beijing, China, June 2008.

[WPO+07] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, K. Asanovic. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, vol. 27, no. 2, March/April 2007.

[WWF+03] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, J. C. Hoe. Applying SMARTS to SPECCPU2000. *Computer Architecture Lab at Carnegie Mellon (CALCM) Technical Report 2003-1*.