SOFTWARE-ORIENTED DISTRIBUTED SHARED CACHE MANAGEMENT FOR CHIP MULTIPROCESSORS

by

Lei Jin

B.S. in Computer Science,

Zhejiang University, China, 2004

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2010

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Lei Jin

It was defended on

Jan 20th 2009

and approved by

Sangyeun Cho, Ph.D., Assistant Professor at Department of Computer Science

Rami Melhem, Ph.D., Professor at Department of Computer Science

Bruce R. Childers, Ph.D., Associate Professor at Department of Computer Science

Onur Mutlu, Ph.D., Assistant Professor at Electrical and Computer Engineering, Carnegie Mellon University

Dissertation Director: Sangyeun Cho, Ph.D., Assistant Professor at Department of Computer Science

ABSTRACT

SOFTWARE-ORIENTED DISTRIBUTED SHARED CACHE MANAGEMENT FOR CHIP MULTIPROCESSORS

Lei Jin, Ph.D.

University of Pittsburgh, 2010

This thesis proposes a software-oriented distributed shared cache management approach for chip multiprocessors (CMPs). Unlike hardware-based schemes, our approach offloads the cache management task to trace analysis phase, allowing flexible management strategies. For single-threaded programs, a static 2D page coloring scheme is proposed to utilize oracle trace information to derive an optimal data placement schema for a program. In addition, a dynamic 2D page coloring scheme is proposed as a practical solution, which tries to approach the performance of the static scheme. The evaluation results show that the static scheme achieves 44.7% performance improvement over the conventional shared cache scheme on average while the dynamic scheme performs 32.3% better than the shared cache scheme. For latency-oriented multithreaded programs, a pattern recognition algorithm based on the K-means clustering method is introduced. The algorithm tries to identify data access patterns that can be utilized to guide the placement of private data and the replication of shared data. The experimental results show that data placement and replication based on these access patterns lead to 19% performance improvement over the shared cache scheme. The reduced remote cache accesses and aggregated cache miss rate result in much lower bandwidth requirements for the on-chip network and the off-chip main memory bus. Lastly, for throughput-oriented multithreaded programs, we propose a hint-guided data replication scheme to identify memory instructions of a target program that access data with a high reuse property. The derived hints are then used to guide data replication at run time. By balancing the amount of data replication and local cache pressure, the proposed scheme has the potential to help achieve comparable performance to best existing hardware-based schemes.

Our proposed software-oriented shared cache management approach is an effective way to manage program performance on CMPs. This approach provides an alternative direction to the research of the distributed cache management problem. Given the known difficulties (*e.g.*, scalability and design complexity) we face with hardware-based schemes, this software-oriented approach may receive a serious consideration from researchers in the future. In this perspective, the thesis provides valuable contributions to the computer architecture research society.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 RECENT DEVELOPMENT OF MICROPROCESSOR TECHNOLOGY

Since the debut of the world's first microprocessor in the early 70's, microprocessor technology has experienced an astonishing evolution. The introduction of the commercial superscalar processors [68] in the late 80's resulted in significant performance improvement. Since then, mechanisms to extract instruction-level parallelism (ILP) have been greatly emphasized in the computer architecture community. Taking the transistor budget for granted, computer architects employed many microarchitecture innovations such as aggressive branch prediction mechanisms, intelligent memory data prefetching, multiple instruction issue, dynamic scheduling, speculative execution, and simultaneous multithreading to keep improving the processor performance [41, 93, 49, 90, 77, 61, 67, 32, 62, 110, 47, 50, 83, 10, 51, 21, 35, 107, 105, 30, 3, 100, 57, 29]. Some of these mechanisms have developed to a level of extreme sophistication and complexity. Luckily, Moore's law [69], which successfully predicted that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years, has been valid for more than three decades. The complexity and inefficiency of these hardware structures have been largely hidden by the fast increase of transistor count on a chip. As a consequence, the microprocessor industry had much success during the last two decades since the release of the first superscalar processor [68].

More recently, the advancement of microprocessor technology has faced some difficulties. Moore's law is challenged by physical constraints and is believed to be impossible to sustain in the foreseeable future [54]. For instance, feature size is being pushed to near physical limitations. Without care, power consumption can grow exponentially. Researchers even predict if the power density is increasing at the current page, the core temperature of the microprocessor is going to trigger nuclear reaction [2]! Another challenge is reliability. With decreased feature size, products become vulnerable to hard and soft faults. The degraded product yield can also increase the cost significantly. Moreover, the margin of performance improvement by using more aggressive speculation is decreasing, even though the hardware design complexity and area budget are increasing exponentially. The fundamental obstacle is the lack of ILP in current single-threaded programs and efficient mechanisms to uncover them. With these issues, superscalar uniprocessor design can apparently no longer sustain the historical performance growth rate. On the other hand, symmetric multiprocessor (SMP) design tries to speed up program execution by running threads on multiple cooperating processors, which are physically separated but wired together through interconnection. However the performance of these workloads suffers from the slow interconnection. As a result, the chip multiprocessor (CMP) architecture [73, 18, 37, 12, 11, 38] is proposed as an alternative solution.

The CMP architecture is like the traditional SMP design. But instead of wiring together physically separated processors, a CMP integrates multiple processor cores on a single die with an on-chip network. In comparison to the SMP design, the on-chip network in the CMP architecture naturally has much lower latency. In addition, the on-chip cache partially replaces the role of main memory and takes the responsibility for synchronizing cooperating cores. Unlike the superscalar uniprocessor architecture that depends on complex pipeline design to extract ILP [91, 75], the CMP architecture is designed to exploit thread-level parallelism (TLP). The individual core design is made simpler in compromise for larger core count. Thus, the design principle here is that the CMP architecture sacrifices performance of individual cores for higher overall throughput. Some studies [98, 99, 70] even propose to use an asymmetric CMP (ACMP) design. In such a design, there is a high performance superscalar core, which provides fast execution of a single-threaded program, while other small parallel cores can run multiple threads simultaneously. By scheduling the workload properly, ACMP is expected to achieve a good balance between performance for single-threaded programs and throughput for multithreaded programs. Based on this discussion, it is clear that the CMP architecture has advantages over the uniprocessor architecture in terms of design complexity, performance scalability and power efficiency.

CMPs with 4 or more processor cores [45, 88, 5, 6] are available now. Some special purpose processors such as Tilera [103] even integrate 64 processor cores or more in a single chip. It is believed that the CMP architecture will become the mainstream architecture. CMPs with hundreds of cores are certainly anticipated.

1.2 CACHE ARCHITECTURES AND WORKLOAD CLASSIFICATION

In company with the shift from uniprocessor architecture to CMP architecture, the memory hierarchy evolves accordingly, mainly adopting ideas from SMP design. To facilitate communication among processing cores, the memory subsystem (off-chip main memory) is shared by all cores within a CMP in one form or another. As a result, the integration of more and more processing cores on a single chip dramatically increases pressure on the memory subsystem [43]. The last-level cache acts as a last defense before letting the request go off-chip. A well-designed last-level cache can relieve pressure on the memory subsystem and off-chip bandwidth. Comparing to the last-level cache design in uniprocessor design, it plays a much more important role in the CMP architecture. Thus designing an effective on-chip cache hierarchy becomes an imminent task for computer architects in the CMP era. In this section, we will first introduce existing last-level cache designs briefly. Then we will discuss common workload types, to which we should pay special attention when designing new cache architectures.

1.2.1 Cache Architectures in CMPs

The research community has paid considerable attention to the last-level on-chip cache. To exploit temporal locality not captured by higher level caches, the last-level cache tends to employ large capacity. The large aggregated footprint size of programs running on CMPs also favors caches with large capacity. However, the amount of die area required is proportional to the cache capacity. The cross-chip diameter increases as the on-chip cache capacity grows, leading to a longer signal propagation delay. The decreased feature size makes the wire delay even worse [13]. In such a case, adopting a traditional monolithic cache structure commonly found in uniprocessors can result in very slow cache accesses.

To improve cache performance, the whole cache space is broken into banks or slices. These banks are connected through wires or on-chip networks [31] to form a logically unified view to the processing cores. The benefits of this approach come in two folds. First, compared with the monolithic cache structure, smaller cache banks have lower access latency. Furthermore, only the accessed cache banks are enabled at a time, providing much better power efficiency. Second, the wire delay can now vary based on the target bank distance. Nearby banks can be accessed much faster due to reduced wire delay even though accesses to farthest cache banks still take the same latency as before. This distance-based latency results in a much lower average access latency than that of the uniform latency design found in traditional cache structure. This improved cache design is called Non-Uniform Cache Architecture (NUCA) [13, 25, 53]. It has a simple design and achieves effective latency reduction. Without significant improvement of on-chip wire delay, NUCA seems an inevitable solution for organizing the last-level on-chip caches with large capacity in CMPs. Moreover, a new CMP design approach [111, 103] becomes very popular recently. The new approach first designs a tile containing processing core, caches, and necessary logic to communicate with others through the network. Then the tile is simply duplicated to meet the core count requirement. This approach can dramatically reduce design effort for large-scale CMPs. The resulting tile-based CMP architecture intrinsically uses a sliced last-level cache design. Thus, NUCA is a natural fit in such a design.

Unfortunately, the speed gap between the processor and the main memory still exists. Effective management of the on-chip cache capacity to minimize the number of expensive off-chip accesses is critical to overall system performance. The introduction of NUCA brings new opportunities for further optimizing cache performance. Since nearby cache banks can be accessed faster than others, placing data close to where they are needed can provide better cache access latency. However, excessive aggregation increases contention in the cache banks, leading to performance loss of off-chip main memory accesses. An intelligent last-level cache management scheme that balances these two conflicting factors to achieve optimal performance is desirable.

The popular last-level cache organizations in CMPs are borrowed from traditional SMPs since they share similarities. These two most widely used designs are the shared cache scheme [45, 6, 46, 56, 71, 84] and the private cache scheme [5, 101, 40]. The shared cache scheme interleaves consecutive cache blocks among all available cache banks based on their block addresses. It provides a logically shared cache view out of physically distributed cache banks. The shared use of all cache banks maximizes cache capacity utilization, thus minimizing the number of cache misses. However, it distributes data purely based on block address, without considering bank distance. A heavily accessed data block coincidentally placed in a nearby cache bank could gain much more benefit than otherwise. For this reason, program performance is sensitive to its cache access pattern and is not predictable to some extent. As CMP core count increases, the performance degradation due to the increased access latency to remote cache banks can become significant. It will eventually out-weigh the benefit of reduced cache misses with large effective cache capacity. Without a significant reduction in wire delay, the shared cache scheme seems not a proper choice for very large-scale CMPs.

The second design is the private cache scheme. Its design philosophy lies on the other extreme of the shared cache scheme. The private cache scheme avoids remote accesses by always keeping a copy of an accessed data block in the processing core's local cache bank. Cache banks are independent of each other and exclusively accessed by their owners. A cache miss in the local cache bank results in a main memory request directly. A directory or snooping mechanism is responsible for maintaining coherence among cache banks. Each processor becomes an autonomous unit, making a CMP easy to scale. Since data lookup is always performed in the local cache bank, wire delay has limited impact on cache performance. However, the strict capacity partitioning often becomes a fatal limitation. In a CMP with N processing cores, the private cache scheme only allows each processor accessing its own cache bank even though the total on-chip cache capacity is N times larger. When a program's working set exceeds the local cache bank's capacity, performance is lost due to conflicting misses, which can be saved with larger cache capacity. If a workload is not distributed evenly, some cache banks can face severe contention while others are barely used. Without caution, the increased number of cache misses can easily offset the benefit brought by the elimination of remote accesses.

Motivated by the limitations of these two basic cache management schemes, many recent proposals try to combine the advantages of both [72, 43, 111, 14, 27, 26, 23, 39]. They can be broadly classified into two basic approaches. One approach starts from a shared cache organization and manages to reduce cache access latency by allowing some degree of replication [111, 14, 26]. The principle is to replicate frequently reused data in a local cache bank so that a later access can be served locally. Replication effectively reduces average access latency. For instance, the victim replication scheme [111] replicates remote cache blocks that are replaced from L1 caches. Instead of replicating all blocks unconditionally as the private cache scheme, the victim replication scheme gives preference to blocks with potential reuse in the future. The other approach is based on a private cache scheme and tries to increase the effective cache capacity by limiting the amount of replication [23]. The suppressed replication of some rarely used data improves cache capacity utilization. For example, the cooperating caching scheme [23] proposes to improve the private cache scheme by assigning high replacement priority to replicated data and globally inactive data. Both approaches seem promising, but challenges remain. First, it is difficult to track program behavior precisely. Without precise run-time information regarding the program's data access pattern, it is difficult to decide the best replication level. Unfortunately, the tracking mechanism often requires complex hardware logic. Second, it lacks a mechanism to inexpensively trace data movement around the cache. In order to maintain cache coherence, any data placement or replication has to be recorded. At cache block granularity, the hardware cost for bookkeeping the related information seems overly expensive.

1.2.2 Workload Classification

Neither the shared cache scheme nor the private cache scheme can perform consistently better than the other because the access pattern varies a lot from one program to another. It is important to have a good understanding of workload behavior before trying to come up with any new scheme that can adapt to different situations. At the fundamental level, all workloads can be broadly classified into three categories: single-threaded workload, latency-oriented multithreaded workload, and throughput-oriented multithreaded workload. These three types of workloads have very distinct behavior. Some of them even have distinctively different goals. For instance, a Fast Fourier Transformation program employs multithreading to speedup the computation. On the other side, a web server application uses multithreading to improve the number of HTTP requests serviced per second rather than the service time of each individual request. In the following, we will examine these three workload categories in detail.

For single-threaded programs, only one thread is active at any time. All data are exclusively accessed by this thread. So no data sharing exists. The workloads in this category are very popular in uniprocessors and still play an important role in CMPs. The access pattern is the simplest among all types, since there is only one data owner all the time. It is always attractive to place data close to where the program is running. But care has to be taken to avoid excessive cache misses incurred by data aggregation. One variation to the single-threaded workload is the multiprogrammed workload. It is composed of multiple instances of single-threaded programs running concurrently. All instances are independent of each other and use separate virtual address spaces. Data sharing among programs rarely happens. Except the interference created by peering programs, the data access pattern is largely unchanged as running alone.

The second category includes multithreaded programs that optimize the execution time. This is the most commonly seen type of multithreaded programs. To improve execution time, the original algorithm in a single-threaded form is rewritten to take advantage of multiple processing elements (*e.g.*, cores in a CMP). A computation task is split into multiple parallel threads. By running threads on spare hardware resources simultaneously, speedup is achieved. A perfect workload partition can lead to an execution speedup proportional to the number of processing cores available in a CMP. However, an ideal speedup is difficult to achieve in real-world applications. Performance scaling is partly hindered by the inter-thread communication, which essentially serializes the execution of participating threads. Furthermore, arbitrary data placement can increase the number of remote accesses, degrading cache performance. If a parallel algorithm is not designed carefully, the benefit of parallelization can be offset by the increased communication cost.

The last group of workloads are throughput-oriented applications. A typical example is a server daemon. In this type of workload, the execution time of an individual thread is still as important as that in a latency-oriented multithreaded program. But the focus becomes how to complete as many transactions as possible in a given time period. Unlike latency-oriented multithreaded programs, some programs in this category can run continuously until terminated explicitly. As a result, the OS virtual memory management policy can have a profound impact on their performance. Moreover, threads in a throughput-oriented multithreaded program are often independent. They communicate and share data among each other, but not in a cooperative way as the latency-oriented multithreaded programs.

Given the various characteristics of these workloads, it is not at all surprising that none of these static cache schemes can serve all of them equally well. Some workloads favor a shared cache scheme, while others may perform better on a private cache scheme. In practical cases, a program can even exhibit different memory access patterns within different execution phases. As a result, a fixed static cache scheme cannot even satisfy the requirement of a single program. A flexible and intelligent cache management scheme that can adapt to different program behaviors is desired in CMPs.

1.3 LIMITATIONS OF EXISTING SCHEMES

Apparently, neither the shared cache scheme nor the private cache scheme alone can suit the memory access patterns of all workloads. Many proposals try to improve the performance of the baseline shared and private cache schemes by adapting to program behavior. They are motivated by different observations and provide distinct solutions. But the basic principles of these proposals are the same: combining the advantages of both shared and private cache schemes to form a new hybrid scheme. Naturally, these architectural innovations are implemented in hardware. Admittedly, a hardware-based solution provides many benefits. First, it hides the underlying detail completely from the OS and the applications. The program behavior tracking and management decisions are performed in parallel with other operations in background. Thus, these designs introduce limited interference to the program execution. Furthermore, the worst case latency of an operation is guarded by the hardware design. The delay is predictable.

However, these hardware-based designs have some limitations: (1) Maintaining up to date information for the executing program is critical to a dynamically adaptive cache scheme. It is not trivial to track program behavior in hardware. The data storage for bookkeeping the event history can become substantial and potentially unmanageable, not to mention the complexity of the control logic. An over-simplified design can generate inaccurate estimation of the current execution status. It can lead to incorrect run-time decisions and impact the program performance adversely. Some statistical sampling method [78] is proposed to reduce the monitoring cost. However, most of these tasks can be easily accomplished by software using off-line analysis; (2) More importantly, these proposed hardware schemes often require a central arbitrator to keep distributed caches coherent. Such a centralized design violates the original purpose of using a distributed cache organization, creating a potential performance bottleneck for large-scale CMP architectures.

1.4 RESEARCH OVERVIEW AND CONTRIBUTIONS

Motivated by these issues in existing hardware-based schemes, **this thesis studies the distributed shared cache management problem using a software-oriented approach**. The proposed solutions are expected to perform at least comparable to available hardware-based cache management schemes, but with much less hardware cost and better scalability.

1.4.1 Problem Overview

In this thesis, we propose a software-oriented last-level cache management approach for optimizing program performance on CMPs. We assume the L2 cache is the last-level cache in a CMP and L2 cache slices are distributed and shared by all cores, similar to recent multicore processors [80, 103, 97]. To remove the constraints exposed in hardware-based schemes, it is important to have a flexible mechanism to locate data in the L2 cache. The underlying mechanism used to control data placement in this work is similar to the page coloring technique [27]. When a page fault is triggered, a free physical page frame has to be mapped to the virtual page generating the fault. During this process, the OS also chooses a target cache slice index and a cache bin number as augmented fields to the page table entry.[1] Whenever a cache block belonging to that page is accessed, the corresponding cache slice index and the cache bin number are fetched to locate it among L2 cache slices. The challenge is to determine a proper cache slice index and a cache bin number for a given virtual page. The proposed software-oriented cache management approach employs a profile analysis method to generate data mapping hints for a program. The hints are then used to guide data placement at run time. The off-line analysis is not limited by hardware resources and can perform a very sophisticated analysis task. By off-loading the cache management task onto software, the proposed approach can exploit a program's cache access patterns that are hard to capture by hardware mechanisms. More importantly, the flexibility of the off-line analysis even allows us to adopt different strategies to tackle programs with various characteristics. In this thesis, a unique strategy is designed for three different types of workloads separately.

For single-threaded programs, it is obvious that distributing data blindly across all L2 cache slices is not an efficient method. If the working set is not large, the remote access delay from the on-chip network can degrade performance. Without causing excessive cache misses, aggregating program working set data as close as possible to where the program runs can provide a considerable performance boost. In this study, the cache access latency (including the network delay) and the cache contention information are considered in combination to compute the cost of a cache access. The derived "optimal" data distribution is the one with minimum total cost for all L2 cache accesses. The evaluation results show that hints are provide up to 3 times speedup with the oracle static 2D page coloring scheme. The average improvement is 44.7%. The dynamic 2D page coloring scheme, which is the online version of the static 2D page coloring scheme, achieves up to 191% better performance than the shared cache scheme with an average of 32% improvement. In addition to the encouraging results,

[1]A cache bin is a collection of consecutive cache sets. The number of sets in a cache bin equals to the page size divided by a cache block size. For example, if a page is 8KB, then a 512KB 4-way associative cache with 64-byte cache block size contains 16 cache bins. Each cache bin has 128 cache sets.

this study also leads to several interesting insights: (1) When NUCA L2 cache is used, cautious data placement is beneficial to single-threaded program performance; (2) In a CMP architecture design, the on-chip network delay becomes a nontrivial performance-limiting factor, which, if not handled properly, will lead to significantly degraded performance; and (3) As the number of cores increases, a flexible L2 cache management framework can be highly beneficial and of growing importance.

Multithreaded programs have much more complex access patterns. Since several threads are active at the same time, data can be requested from multiple sources simultaneously. It is not obvious immediately which is the optimal location for the data. However, it is observed that the data access behavior of a program is closely related to its intrinsic characteristics. For instance, an array is accessed by an index. In a sophisticated program, the index is usually computed from other variables, such as thread ID and loop iterators. Even though the values of these dependent variables can change, the way the index is calculated is determined by the algorithm itself. As a consequence, its access pattern persists. To gain basic understanding of the cache access behavior of multithreaded workloads, profile analysis is carried out. Some observations from the study are very motivating and worth mentioning.

Figure 1 shows the data sharing behavior of *cholesky* from the SPLASH-2 benchmark suite [86]. *Cholesky* is a representative latency-oriented multithreaded programs. In the figure, both curves represent the cumulative percentage of accesses to data pages that are shared by different number of threads. But the number of sharing threads is counted differently for these two curves. For the dark blue curve (diamond), the number of sharers for a page is counted over the entire program execution. For the light green curve (square), the number of sharers for a page is determined by the maximum number of simultaneous sharers at any instant during the program execution. As the figure shows, nearly 17% of the all accesses are shown to go to pages predominantly accessed by a single thread. At the top right of the plot, it shows around 40% of all accesses touch pages that are shared by all threads. This type of curve shape is common in SPLASH-2 programs, suggesting that many data pages are either private or highly shared. Indeed, this is intuitive as scientific workloads often involve partitioned data processing (private data) and global synchronization and data exchange (highly shared data). For private data, it is desirable to place them in the requester's

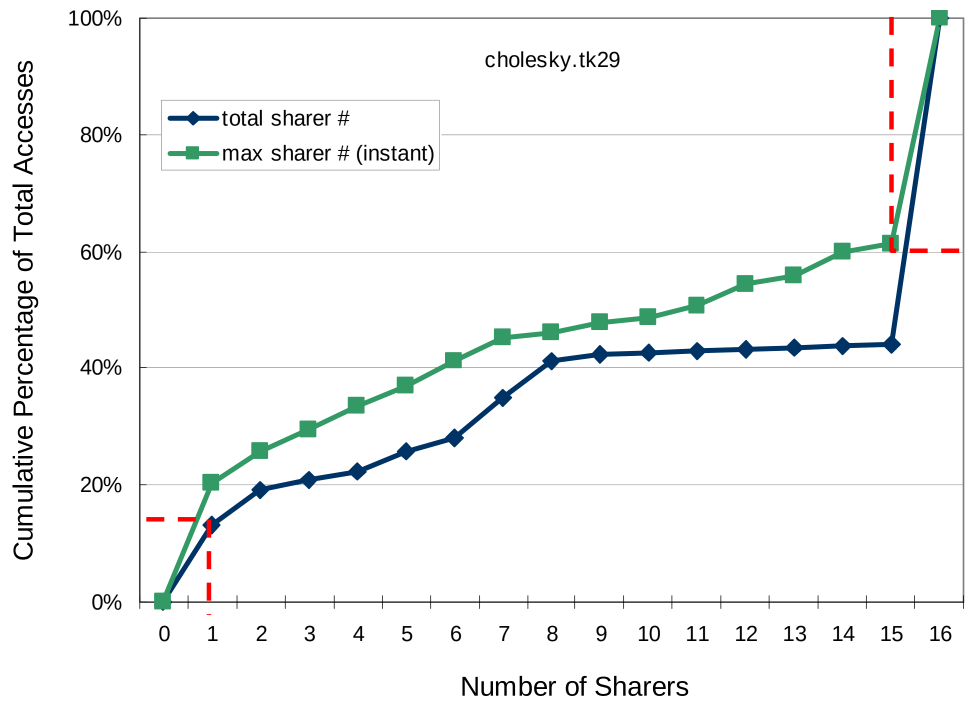


Figure 1: Data sharing behavior of cholesky at page granularity. The two curves capture the number of memory references to a page shared by a different number of threads. For the dark curve (“total sharer #”), the number of sharers is determined by counting the total number of threads accessing the page over the entire program run. For the light curve (“max sharer #”), the number of sharers is determined by recording the maximum number of simultaneous shares for that page at any instant during the program run.

local cache slice at the earliest to avoid remote accesses. On the other hand, it is beneficial to replicate highly shared data or fetch them from neighbors nearby. This observation uncovers an excellent data locality optimization opportunity by using the software-oriented shared cache management approach. In more detail, the compiler can employ a K-means clustering based algorithm to classify data access patterns for latency-oriented multithreaded workloads. The generated hints, which are independent of program inputs and cache configurations, are opportunistically used by the system to improve program performance. The experimental results show that the proposed scheme improves the evaluated multithreaded scientific program performance by 10% over the shared cache scheme and achieves similar performance to the victim replication technique [111]. When data replication is enabled and guided by the hints, the proposed scheme brings additional 9% performance gain, performing 19% and 9% better than the shared cache and the private cache respectively.

The throughput-oriented multithreaded programs have very different nature in comparison to the latency-oriented multithreaded programs. Thus a new strategy has to be taken. However, it is difficult to extract access patterns of a throughput-oriented multithreaded program through off-line trace analysis. Due to the randomness of the incoming requests and the impact of thread scheduling by the OS, this type of program may even have no trackable patterns at all. We have not found any related work from the literature in this aspect. In this study, we take an indirect approach to characterize the data access behavior of a program. Instead of analyzing accessed data directly, we examine the memory instructions which access those data. The profile study shows that the reuse property of a memory instruction is quit persistent. That is, a data block accessed by a memory instruction exhibits frequent reuse during its residence in the cache, then other data blocks accessed by the same memory instruction tend to have similar reuse pattern. As a result, the reuse statistics of memory instructions from a profile analysis can be used to predict data reuse patterns. The data reuse information is a good hint for controlling data replication in NUCA caches. For example, if the data touched by a load instruction exhibit good temporal locality during a profile study, it is beneficial to always allow replication of the data touched by this instruction based on our observation. Thus, this load instruction itself becomes a hint for data replication. Here, there is no need to track data addresses as hints for online execution. In addition, this scheme is not limited to throughput-oriented programs. The idea is general and can be applied to other types of workloads as well. The evaluation results show that the proposed hint-guided data replication control scheme is effective in locating the optimal data replication level, achieving performance that is comparable to the best of other compared cache schemes.

1.4.2 Research Contributions

This thesis studies a software-oriented shared cache management approach, which is considered as deviating from conventional hardware-based cache management schemes substantially. It opens up a new research direction for the last-level cache management. To the best of our knowledge, this is the first study for software-oriented NUCA cache management strategies. Moreover, the proposed software-oriented approach is orthogonal to other hardware-based schemes and can be used in combination.

Through the study of the proposed software-oriented schemes, this thesis makes the following contributions:

- A page-granularity data mapping mechanism is proposed. By transferring the data placement control to virtual memory management, the proposed mechanism offers a novel way to manage traditional shared caches through software.
- Based on the page placement mechanism, three different optimization strategies are proposed for single-threaded programs, latency-oriented multithreaded programs, and throughput-oriented multithreaded programs respectively. They are evaluated and compared to the shared cache scheme, the private cache scheme, and their variants. The experimental results show that the proposed schemes are effective for boosting the performance of the baseline shared cache schemes, achieving better performance than other compared schemes.
- For single-threaded programs, static and dynamic 2D page coloring schemes are proposed. They improve a previous cache optimization scheme [82], which focuses on cache conflicts removal. To adapt to the CMP architecture, the proposed schemes try to balance both cache access latency and cache miss rate on a NUCA.
- For latency-oriented multithreaded programs, a novel cache access pattern recognition algorithm is proposed. By utilizing machine learning techniques, the recognized patterns are effective across different program inputs and cache configurations. The derived data affinity hints can guide data placement at run-time for improved data locality.
- For throughput-oriented multithreaded programs, a novel profile-guided data replication control scheme is proposed. Through off-line analysis, a set of memory instructions

that contribute to most of the data reuse are identified. This information is then used to guide online data replication. It is worth mentioning that there are only few profile-based studies for this type of workloads.

1.5 DISSERTATION ORGANIZATION

In the remaining chapters of this thesis, previous work and related background are first presented in Chapter 2. In Chapter 3, the machine model on which the proposed schemes are based and the detailed experimental setup are provided. Then the static 2D page coloring technique and its dynamic counterpart are introduced in Chapter 4, along with the evaluation results. Chapter 5 first analyzes common memory access patterns found in latency-oriented multithreaded programs. Based on the analysis, a novel pattern recognition algorithm for generating data affinity hints is introduced. Then the evaluation results are given. Chapter 6 analyzes the obstacles to perform profile-assisted optimizations for throughput-oriented programs and common characteristics of these programs. Then it introduces the profile-guided data replication control scheme, followed by evaluation results. Finally, the summary of this thesis work and the future research plan are described in Chapter 7.

2.0 BACKGROUND AND RELATED WORK

In this chapter, we review related background and previous work in detail. First, we delve into the basic shared cache scheme and the private cache scheme, as well as point out the pros and cons of both. Then, previously proposed hybrid cache management schemes that try to combine the advantages of the shared cache scheme and the private cache scheme are discussed. This thesis work also bears some similarity to the memory management schemes in Cache-Coherent Non-Uniform Memory Architecture (CC-NUMA) or Cache-Only Memory Architecture (COMA) machines [41]. The related literature is introduced at the end.

2.1 HARDWARE-BASED CACHE MANAGEMENT SCHEMES

Almost all cache schemes proposed in recent literature or adopted by products in industry are purely hardware-based schemes. In these schemes, the cache controller manages where and how a block of data should be placed using predetermined rules. At the time of a data access, the cache controller seeks data by following the same rules and supplies the data to the processor pipeline. Some of them may require auxiliary hardware components, but all hardware-based schemes introduced in the following sections essentially differ only in these rules about data placement and replacement.

2.1.1 Shared Cache Scheme

There are two baseline L2 cache management schemes in the current-generation of CMPs: the *shared cache scheme* and the *private cache scheme* [5, 6, 45, 40, 46, 56, 71, 84, 101].

The shared cache scheme is a natural evolution from the cache structure of uniprocessors. However, a monolithic shared cache structure is not suitable for CMPs. It is often split into banks or slices to reduce cache access latency. All banks or slices are connected together through wires or networks to form a logically unified cache space. The whole cache space is shared and accessible by all cores in a CMP. If a shared cache is composed of multiple cache slices, the common approach is to interleave consecutive memory blocks across available cache slices based on their block address [6, 46, 56, 71, 84]. Spreading data in this way helps balance cache accesses among all slices, mitigating hot-spots and contention. By aggregating all cache slices, the shared cache scheme can utilize the cache capacity effectively. As a result, it leads to a low cache miss rate, which is often the most important feature when designing a cache for a uniprocessor. However, the biggest disadvantage of the shared cache scheme for a CMP is that accesses to remote cache banks or slices may incur considerable delay on the wires or interconnection network. In certain situations, data are blindly distributed to remote slices even though the nearby cache slices are underutilized. This causes an unnecessary latency penalty and negatively impacts the cache performance.

2.1.2 Private Cache Scheme

The private cache scheme adopts a different philosophy as compared to the shared cache scheme. It divides the whole cache space into private cache partitions (slices), each of which is owned by one core in a CMP [5, 40, 101]. A core has exclusive access to its own private cache slice. The cache is often arranged in a way such that a core has the shortest distance to its corresponding cache slice. On a cache access, only the local cache slice is searched. A cache miss in the local slice results in a memory request. The cache coherence mechanism then determines if the requested data should be fed from the off-chip main memory or a neighboring cache slice. A returned data block is always replicated in the local cache slice in the hope that the next access to the same data block can be resolved locally. If this is the case, a slow remote cache access in the shared cache scheme is turned into a faster local access in the private cache scheme. The latency saving can lead to better cache performance than the shared cache scheme if a cache request sequence exhibits sufficient reuse patterns.

Given the diversity of workloads characteristics, however, this does not always happen. Since the whole cache capacity is split into N partitions for a N-core CMP, the effective capacity seen by each core in the private cache scheme is much smaller than the shared cache scheme. As a result, the cache miss rate becomes more sensitive to the program working set size. So the private cache scheme often causes higher cache miss rate than the shared cache scheme. When the cache miss rate increases, the benefit of local cache accesses starts to shrink and is finally offset by cache miss penalties (the more expensive off-chip main memory accesses). For this reason, the private cache scheme is not the best design for all workloads.

2.1.3 Hybrid Cache Schemes

It is apparent that neither the shared cache scheme nor the private cache scheme can perform consistently better than the other for all workloads [72, 43, 111, 27]. For example, streaming programs with little temporal locality or programs with their working sets entirely fit into a cache slice can perform better with the private cache scheme, while programs with large working sets or a high degree of read-write data sharing may perform better with the shared cache scheme. Researchers have thus examined many possible solutions. The basic principle of those proposals is to combine the advantages of both the shared cache scheme and the private cache scheme, while avoiding their shortcomings. This goal is achieved by balancing between latency (by improving data locality) and cache miss rate (by utilizing cache capacity more efficiently).

Speight *et al.* [85] presented an eight-core CMP architecture design where each two cores form a cluster and share a private L2 cache. Within an L2 cache cluster, four cache banks are shared, providing fast access to shared data among the two cores. But each L2 cache cluster is a private cache from the viewpoint of the CMP chip. So coherence should be maintained among the four clusters. When a cache line is evicted from an L2 cache cluster, hints are consulted to decide if the victim should be written to the off-chip main memory or kept in a peer L2 cache cluster. If the victim is kept in a peer L2 cache cluster, an L2-to-L2 transfer is much faster than an off-chip main memory access when it is reused. Hints are constructed dynamically in tables within each L2 cache cluster during a program execution.

The performance of this scheme depends on the design-time parameters such as the cache size and the number of cores in a cluster. Grouping threads with sharing data into a cache cluster would provide much benefit. However it may be hard to predict the performance of an application developed without any knowledge of the parameters unless the OS makes informed process scheduling decisions.

Similarly, Huh *et al.* [44] proposed an L2 cache organization with a configurable sharing degree. To be more specific, it clusters L2 cache banks based on a configuration set by the OS. Cores assigned to an L2 cache bank cluster have exclusive access to it. By varying the degree of clustering, this scheme essentially trades off between purely shared cache scheme and purely private cache scheme. They also studied a static cache mapping policy and a dynamic cache mapping policy. The dynamic cache mapping policy allows a cache block to go to multiple candidate banks. With proper placement of a cache block, its access latency can be improved. Furthermore, with the dynamic mapping policy, a generational promotion algorithm can be used to guide cache block migration, which can further improve the access latency depending on the cache block usage. The more cache banks it allows to go to, the more aggressive the optimization becomes. However, an L1 cache miss can trigger a search in multiple cache banks in parallel. Even though partial tags are added to help reduce the pressure of full tags, this can still consume substantial cache bandwidth. Thus the search cost largely determines the freedom of the dynamic mapping policy. This scheme is not considered to be scalable for large-scale CMPs.

Zhang and Asanović [111] proposed the *victim replication* scheme based on a shared L2 cache organization. In the proposed scheme, each L2 cache slice can replicate a data block from a remote cache slice when it is replaced from the L1 cache. The replication saves accesses to remote L2 cache slices or directories when the replicated data blocks get reused in the future. This scheme can boost shared cache performance significantly when the program's working set fits well into its local L2 cache slice, bringing the performance close to that of the private cache scheme. Essentially, the local L2 cache slice provides a large victim cache space for the cache blocks whose home are remote so that data locality can be improved. However, a coherence request or an L1 cache miss calls for more actions than before, because locating a cache block is no more deterministic. Both L1 and L2 caches (in parallel or in sequence) should be checked upon an invalidation request because it is not readily known if a remote cache block has been replicated in the local L2 cache slice. The local cache slice must be always searched on an L1 cache miss regardless of the missed address. As a result, cache pressure is increased substantially. Further, the replicated victims can kick out unowned cache lines in the L2 cache. This unwanted consequence can increase the number of cache misses considerably, incurring expensive off-chip main memory accesses. Under certain circumstance, victim replication becomes too aggressive and degrades program performance.

Beckmann *et al.* [14] proposed a controlled victim replication scheme called *ASR*, which tries to reduce cache pollution caused by excessive replication in the original victim replication scheme. ASR employs a technique called *selective probabilistic replication* to achieve controlled replication of frequently reused data. To measure the best replication level, their design introduces a set of tables along with each core to keep track of the performance gain and loss with regard to increasing or decreasing replication level. By adjusting the replication level adaptively during a program execution, their scheme is shown to achieve the near-optimal balance between the miss rate and access latency. However, these tables take considerable effort to maintain. The control logic can also be very complex. These hardware structures can consume a lot of chip area. If such a complex scheme is suitable for a practical design is unclear. Furthermore, the selective probabilistic replication technique relies on a ring network to merge a victim with a copy in a remote cache slice. This limits it from scaling up to a large number of cores.

Chishti *et al.* [26] proposed a cache scheme called *CMP-NuRAPID* having a hybrid of private per-core tag arrays with forward pointers and a shared data array with reverse pointers. By utilizing those pointers, cache blocks can virtually reside in any L2 cache slices. Based on this hardware mechanism, they studied a series of optimizations, such as controlled replication, in-situ communication, and capacity stealing. The controlled replication duplicates reused data blocks in L2 slices nearby to avoid remote cache accesses. The in-situ communication optimizes the access latency of read-write shared data by placing data blocks closer to readers and updating the data blocks on every write actively. This allows direct data block reads without incurring coherence misses. The capacity stealing technique allows a core with large capacity demand to demote its less-frequently-used data to neighboring L2 slices. By using this technique, a core can attract as many data blocks as necessary to improve data affinity without suffering from the limited cache capacity as in the private cache scheme. However, the flexibility of data movement among L2 cache slices does not come as free. First, CMP-NuRAPID doubles the size of the tag array, incurring storage overhead. Second, the modification to the coherence protocol increases the complexity of the control logic substantially. Lastly, the use of bus and broadcast policy for making private tag arrays consistent limits its capability to be adopted in large-scale CMPs.

Chang and Sohi [23] proposed a *cooperative caching* framework based on the private cache scheme via cooperative actions among cache slices. They studied optimizations such as cache-to-cache transfer of clean data, replication-aware data replacement, and global replacement of inactive data. The cache-to-cache transfer of clean data is a natural extension to the traditional cache coherence design in a multiprocessor system as the network latency becomes much lower in CMPs. The replication-aware data replacement overrides the LRU policy and gives the eviction priority to replicas in a private cache. This technique intends to overcome the problem of excessive replications in the private cache scheme. The global replacement of inactive data breaks the rigorous partitioning of a private cache space by allowing local L2 cache victims to replace inactive cache blocks in peer L2 cache slices. However, all those cooperations among multiple private L2 cache slices are implemented assuming there is a facilitating data structure called Central Coherence Engine (CCE). The CCE maintains information regarding replication status of all cache blocks. It has to consult the CCE on every optimization decision and status update. As admitted in the paper, this centralized structure becomes performance bottleneck when the number of cores is beyond 8.

Michael and Mark [65] proposed a two-level cache coherence based on the conventional shared cache scheme for CMPs. In their scheme, each program is assigned a home node table, which maps lower bits of a data block address to a target home node. By carefully designing the mapping, all accessed cache blocks of a program can be grouped in cache slices nearby where the program runs. The first-level cache coherence is responsible for maintaining consistency among these L1 caches and the corresponding home nodes. By grouping related threads together and binding home nodes tightly, the average access latency to these home nodes through the first-level cache coherence becomes lower than the shared cache scheme, which distributes data to all available cache slices blindly. On an L1 cache miss, a lookup is performed in the home node. A miss in the first-level cache coherence can involve extra directory access in the second-level cache coherence. If the data is not found in the home node, a request is sent to the directory like the conventional cache coherence. The directory then either broadcasts or directly notifies the owner to forward the data. This scheme essentially clusters data closely to the threads of a program by using a flexible mapping mechanism. The focus of this paper is to optimize the cache performance of large-scale CMPs for multiple co-scheduled server programs. No effort is made to improve data locality within a cluster. As a result, it only benefits when the number of threads is smaller than the number of available cores within a CMP.

Dybdahl and Stenstrom [33] proposed a hybrid scheme which divides each core's local cache slice into shared and private partitions, assisted by a modified replacement policy. The shared partitions from all cores form a unified shared space. The *Sharing Engine* estimates the best partition size for private partitions based on the augmented per-set *shadow tags* and the LRU information. An accessed data block is first allocated in the local private partition, which is always accessed first on a data request. An eviction from a private partition replaces a cache block in the shared partition. The candidate for replacement is determined based on the partition estimation and the LRU information. A miss in the private partition leads to parallel search on the local and neighboring shared partitions. Their scheme provides the benefit of fast access of the private cache scheme while loosens the cache capacity constraint to include neighboring cache slices. The evaluation results show that this smart partitioning scheme performs better than both the private cache scheme and the shared cache scheme as expected. It also outperforms the *cooperative caching* scheme proposed by Chang and Sohi [23]. However, the centralized Sharing Engine for estimating cache partition size, controlling shared partitions, and enforcing the modified replacement policy is not scalable. As a result, the proposed scheme suffers the same limitations of the other related hardware-based schemes.

2.2 PAGE-BASED CACHE MANAGEMENT

The page coloring technique was first proposed by Kessler and Hill [52] to reduce cache conflict misses. They derived a simple model that shows a naive page placement policy can lead to up to 30% more unnecessary cache conflicts. Based on this observation, they developed several page placement algorithms that reduce cache misses by 10-20%. The proposed page coloring policies are relatively simple. They color pages based on static heuristic rules. So it still leaves a lot of space for further improvement. Romer *et al.* [87] studied several dynamic page mapping policies for reduced cache conflicts. They proposed two types of mapping policies. The active policies closely monitor the TLB contents and proactively remap a page that has potential to cause conflicts to a different color. The other group, called periodic policy, is more passive and only periodically checks the TLB for selecting any pages that violate the rules as recoloring candidates. Similarly, Bershad *et al.* [15] used cache miss lookaside buffer to detect cache conflicts by recording and summarizing a history of cache misses. Based on these hints, the OS virtual memory management module plays the role for remapping a page that suffers from a large number of conflict misses.

Alternatively, Bugnion *et al.* [20] proposed a compiler-directed page coloring technique. Their technique utilizes the compiler's knowledge of the access patterns of a parallelized application to reorder and group the pages that are accessed by the same processor together. Then these pages are colored cyclically to avoid conflicts based on the principle that pages accessed by the same processor should be spread to different colors as even as possible. Also arrays accessed by the same processor should be assigned with different starting addresses. The derived coloring plan guides memory page mapping in the OS at run time.

Sherwood *et al.* [82] proposed a profile-based method to guide page coloring for reduced conflict misses. Their static approach uses a heuristic algorithm to analyze an L2 cache access trace and estimates the number of potential conflicts between any pair of pages. Given the inter-page conflict statistics, an optimal page placement is arranged to achieve minimum number of conflict misses prior to a program execution. The derived hints then guide page placement at run time for the same program and input set. They also propose a dynamic scheme, which tries to recognize the pages that are responsible for the high conflict misses in a cache bin and migrate them to a different one. The evaluation results show that the dynamic scheme is very effective in reducing the number of expensive main memory accesses for a unified L2 cache. However, these early works all target for the cache architecture of uniprocessors. The problem caused by non-uniform cache access latency in CMPs is not considered.

There are several other works inspired by the page coloring based cache management framework. Instead of using a simulation-based evaluation approach, Lin *et al.* [63] modified memory management module in the OS to support different page coloring policies. Then they evaluated in a real machine several previously proposed cache partitioning policies, which target for different goals, such as performance, fairness, and quality of service (QoS). Through the study, they confirmed several important conclusions from the prior work and obtained new insights that are unlikely to get from a simulation. However, only single-threaded programs and multiprogrammed workloads are studied in this work. Multithreaded scientific programs and server programs can exhibit distinct cache behavior and require a different cache policy. Besides that, the cache partitioning scheme only improves the cache miss rate by isolating the impact from other co-running programs, it does not improve the data affinity for the non-uniform latency cache at all.

Chaudhuri [24] proposed a hardware-based page migration scheme. His proposal is motivated by the observation that a majority of pages are only accessed by a single core during consecutive sampling periods. These "solo pages" often exhibit good locality. If a page is accessed by a core within a sampling period, it is very likely that most of the accesses to that page would come from the same core in that epoch. His scheme adopted a simple threshold-triggered page migration policy. The migration destination is determined based on the principle of minimizing the average access latency for all sharing cores. The evaluation results show significant improvement over the conventional shared cache scheme. However, the tracking of a page usage and its location by hardware poses a big challenge. The solution given in this work employs a set-associative page access counter table to measure the usage. It also uses a mapping table in both L1 cache and L2 cache to track the page movement. Even though the author claims that the storage overhead of these tables is only 4.8%, the logic required to maintain these tables becomes intractable, let alone the complexity of the migration protocol. Furthermore, it requires special care to maintain table consistency and to avoid deadlock. The complexity also makes the job of logic design and verification overwhelming.

Awasthi *et al.* [9] proposed a more practical page migration scheme. By modifying the TLB and utilizing unused bits in the original page address, it allows a main memory access with the original physical address and an L2 cache access with a manipulated address. Decoupling the cache access address and the main memory access address makes page migration scheme very flexible. The scheme enables the OS to manage cache capacity allocation to all cores within a CMP by dynamically migrating pages at the end of each epoch. The goal of capacity allocation is to minimize the cache miss rate. They also studied a policy to control the migration of shared pages. Based on the usage, a page is migrated to a location where the total access latency of all sharers is minimized. Their evaluation results show 10% to 20% performance improvement compared to the shared cache scheme. However, it is not very clear from the paper how the page migration process is done and how the cache coherence is maintained during the migration. Another concern is that the page migration scheme can suffer the "ping-pong" problem as discovered in the previous data block migration schemes. Many evenly shared pages may end up residing in these central cache banks, creating hot-spots.

Finally, Hardavellas *et al.* [39] analyzed L2 access traces of scientific and server workloads. They observed that the L2 cache accesses can form three categories: (1) Highly shared read-only instruction accesses; (2) Mostly read-write shared data accesses; and (3) Private data accesses. They introduced a new address interleaving mechanism so that data can be clustered around the accessor in a controlled manner while still retaining quick lookup within the cluster. Besides, they proposed to optimize access categories differently at page granularity. First, instruction pages are replicated and spreaded nearby the requesting core. Then data pages are initially classified as private and loaded into a local L2 cache slice. If a data page is detected as actively shared, the corresponding TLB and related cache contents are invalidated. The page is then distributed cross all cache slices using the traditional address interleaving method. By differentiating read-write shared data from private data and read-only shared data, it enables data replication without incurring cache coherence problem. As a result, their scheme avoids the hardware complexity introduced by previously proposed data replication and migration mechanisms. However, one obvious shortcoming of this scheme is that the classification conversion cannot be reversed. Once a page is deemed as shared, it is distributed like the conventional shared cache scheme until the end of the program execution. For long-run programs, especially server workloads, this undesirable feature can gradually degrade their scheme to the traditional shared cache scheme.

2.3 SCHEMES FOR NON-UNIFORM MEMORY ARCHITECTURE

The problem of tackling non-uniformity of the L2 cache access latencies (*i.e.*, NUCA) bears similarity to the problem of attacking disparate memory access latencies of the distributed shared memory in a SMP, such as the Non-Uniform Memory Architecture (NUMA) or the Cache-Only Memory Architecture (COMA) machines [41]. Because the ratio between local memory accesses and remote memory accesses largely determines program performance in such a machine, it is of utmost importance to improve the data affinity at the level of distributed main memory by carefully placing, migrating, and replicating pages [42, 17, 28, 59, 22, 106, 94, 108]. Indeed, NUMA is similar to the shared cache scheme while COMA behaves like the private cache scheme. Previous studies of these schemes provide valuable insights.

Holliday [42] studied the paged main memory management in a local/remote architecture for shared memory multiprocessors. He addressed issues regarding the architectural support for recording page reference history, the impact of a page size, and the operating system support for page migration.

Bolosky *et al.* [17] proposed a very simple page placement policy, which initially places a page in the local memory and migrates it to other processor's local memory as requested. A page is put into global memory after the page movement passes a threshold. They defined three statuses for a page: read-only, local-writable, and global writable. The policy decision combined with a page status decide the action to perform for a memory request. They found the simple page placement policy performs well and achieves near optimal performance.

Verghese *et al.* [106] classified all memory pages to three categories, which are very similar to the Hardavellas' classification [39]. Based on these page categories, they devised a decision tree to control page migration and replication in the operating system through monitoring some event counters. The evaluation results show that it achieves up to 30% improvement over the ordinary NUMA scheme. Wilson and Aglietti [108] extended Verghese's work and further studied a dynamic page placement policy for a TPC-C workload.

Soundararajan *et al.* [94] studied the data locality property and performance of the baseline NUMA scheme, the NUMA with remote access cache (RAC), the COMA, and the NUMA with page migration and replication. Through experiments, they found RAC can effectively capture short-term temporal locality of fine-grain data. Also the page migration controlled by the OS can adapt to program memory usage (long-term data sharing at page-granularity) and move a page's home to the local memory. As a result, they proposed to combine these two techniques so that they can work in a synergistic way. The evaluation shows that the new scheme performs well and is robust.

Chandra *et al.* [22] studied the OS process scheduling policy for improving data locality of a CC-NUMA machine. To increase the affinity of data that already reside in memory, the modified OS scheduling manager tries to resume the execution of a sleeping process on a processor or a processor cluster where it has been running before switched out. On the other hand, they also employed a page migration policy, which uses a TLB miss as the signal. This scheme is simple and effective.

Marathe and Mueller [64] proposed a hardware profile-guided page placement scheme for reducing memory contention. In their work, a truncated version of code is profiled each time before a program is about to execute. The sampled memory access trace is then used to determine the optimal placement for each touched page. Then the derived affinity hints are used to direct page placement in the full program execution. This method is effective in improving data locality but has several limitations. First, it requires to profile the memory access trace every time before a program's execution. The quality of the truncated code is crucial to the derived affinity hints. Generating representative code for the whole program is difficult. Furthermore, they assume that a dynamic memory allocation returns the same address for both the profile execution and the full program run. This is not always true and could hurt program performance significantly when the assumption does not hold.

Tikir and Hollingsworth [104] studied the data locality property of the *specjbb* server benchmark on a Java virtual machine and found that different regions in the Java virtual machine heap have unique characteristics. This observation led to a segmented heap design for young generation objects, where each segment is allocated to a processor's local memory. On the other hand, objects in an old generation region of the heap survive multiple garbage collections and can be accessed by different processors in different intervals. They propose to allow migration of those long-life objects to improve the data locality. The evaluation results demonstrate the effectiveness of their scheme.

Besides those experimental approaches, there are also a lot of theoretical studies on this topic [16, 55, 1, 34, 36]. They tried to model the complexity of a system with non-uniform access latency. Based on the model, the performance bound of all kinds of page replication or migration algorithms were derived mathematically. Some even proposed online algorithms that do not require oracle information to achieve performance within a given bound of the optimum.

As seen in this section, previous works for NUMA systems put a great emphasis on the data locality issue as they all employed migration policies to bring frequently accessed data closer to where they are needed. This is because the remote access delay can be very severe in these systems where processors are physically separated and connected through networks. The delay on the networks and the latency of a local memory access can differ by up to a magnitude. On the other hand, the main memory capacity is relatively large. Given a program with a reasonable working set size, the contention caused by limited memory capacity may not be significant. As a consequence, reducing the memory contention is not as urgent as reducing the remote access latency. However, all processing cores are co-located in a single CMP chip and communicated through on-chip networks. The remote access is much faster than that in a shared memory multiprocessor. But the limited on-chip cache budget indicates an efficient contention management is very important. Due to the shift of trade-offs between the access latency and the storage capacity as we move from SMPs to CMPs, the performance loss incurred by cache contention cannot be ignored.

The cache miss rate was the factor with utmost concern when optimizing the cache performance in the uniprocessor era. In a CMP with NUCA caches, it becomes ineffective to only consider the impact of access latency and cache miss rate separately. Furthermore, data in memory are managed at page-granularity naturally because of the virtual memory management. Page table is all it needs to keep track of data movement and maintain coherence. In cache, however, data are loaded and stored at cache block granularity. Migrating data blocks around imposes a big challenge on the cache coherence module. Tracking data location becomes too expensive to handle. Managing data at a larger granularity, such as a page size, can certainly mitigate this problem. But the migration cost that comes from invalidation and data copying becomes undesirably large. The last-level cache is higher in the hierarchy than the main memory. Thus its performance plays a more critical role than the main memory in terms of the whole system. From this perspective, effectively managing NUCA caches in a CMP is much more difficult than the task in NUMA, even though they share a lot of similarities.

3.0 MACHINE MODEL AND EXPERIMENTAL SETUP

3.1 MACHINE MODEL

Throughout this thesis, we assume a tile-based CMP architecture as shown in Figure 2. Examples of the tile-based organization include the TILE64 processor from Tilera Corporation [103] and the 80-core prototype Polaris from Intel Corporation [81]. Many research proposals also adopt similar architectures [111, 65]. The core count within a CMP can range from two to over one hundred. Several factors prohibit the adoption of a very aggressive configuration in this work: (1) A simulation-based experimental method is used for this thesis. While choosing an aggressive CMP configuration may exaggerate the benefit of the cache management schemes proposed in this thesis, simulating a large number of tiles can be prohibitively slow; (2) On the other side, the CMP configuration has to be large enough so that problems with remote access delay can be clearly demonstrated. Therefore, we assume a CMP has 16 tiles in this study. Since the proposed software-oriented approach does not suffer the bottleneck in hardware-based schemes, we believe the conclusions drawn from this study can be applied to other larger scale CMPs as well. Within a CMP chip, the 16 tiles are organized as a 4 x 4 grid, connected through an on-chip 2D mesh network. Each tile consists of a core, a private L1 instruction cache, a private L1 data cache, and a slice of globally shared L2 cache. Cache coherence is maintained by the directories distributed along with L2 cache slices. A tile is connected to the on-chip network with a 5-way switch.

To simplify the study in this thesis, we pin each thread to its specific core and assume this binding does not change during the trace collection stage and the actual performance measurement stage. In practice, the data placement should also consider the impact of the thread scheduling. This is beyond the scope of this thesis.

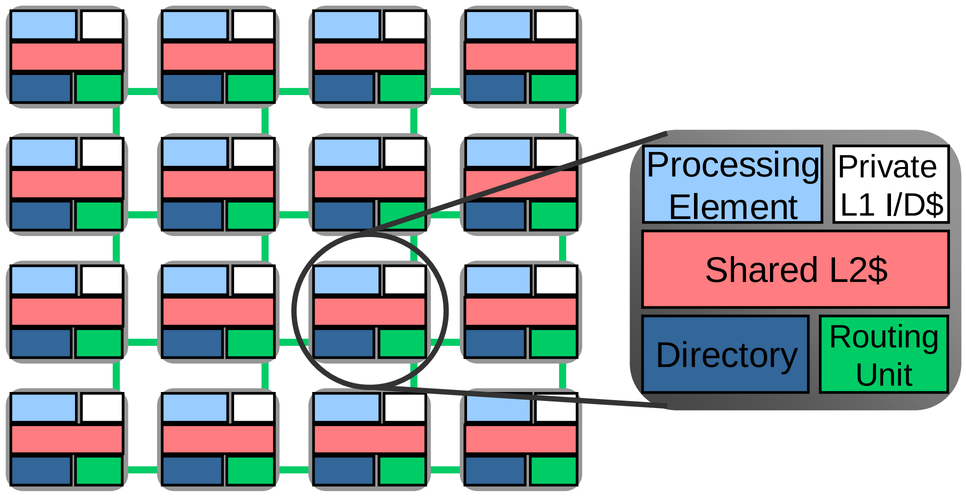


Figure 2: A tile-based CMP architecture organized as a 4 x 4 2D mesh. Tiles are connected through on-chip networks.

The traditional page coloring technique controls data placement in a physically addressed cache by choosing a proper physical page at page mapping time. While this process can be done easily with a slight modification to the OS page fault handling routine, the coloring range can be limited by the availability of free physical pages in the page list. As an alternative, we extend the previously proposed TLB-based mechanism [82] to allow flexible management of page placement in a L2 cache. As illustrated in the Figure 3, two extra fields named tile ID (*TID*) and cache bin index (*BIN*) are attached to the ordinary page table and TLB entries. A main memory access still uses the translated physical address as before. The address for the L2 cache access needs special handling. Instead of using bits from the original physical address to index a cache, the values in *TID* and *BIN* are used to locate a cache slice and a cache bin within that slice. This essentially allows a page to be placed on any cache bin by defining the values for *TID* and *BIN*. The address for a main memory access is decoupled and does not constrain page placement in the cache. In addition, both *TID* and *BIN* can have "null" value, commanding data distribution at cache block granularity for the current page just like the conventional shared cache scheme. In this way, the two addressing modes can co-exist in a single scheme. Depending on the access pattern of a page, one of

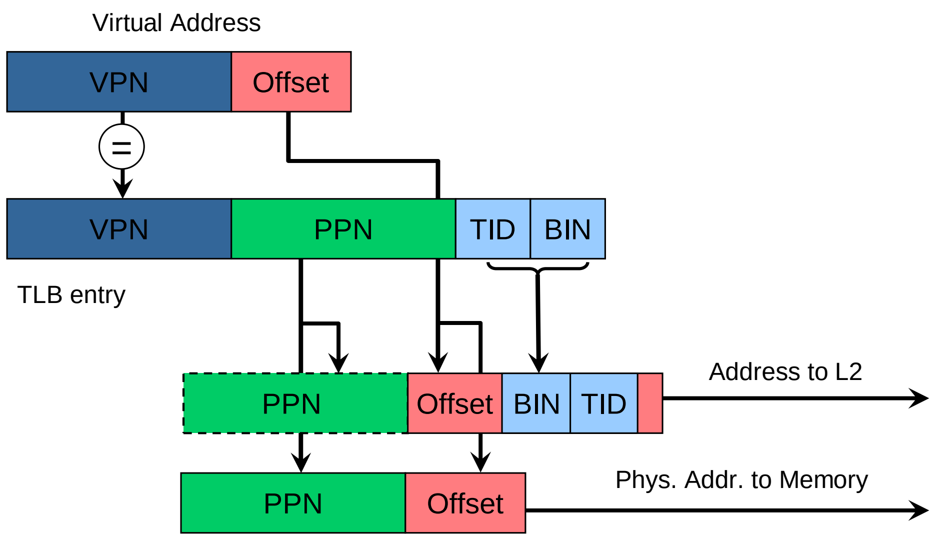


Figure 3: A TLB entry is augmented with tile ID (TID) and cache bin (BIN) fields. These two fields together with higher bits from a page offset are used to index the L2 cache. The L2 cache tag field is extended to accommodate the whole physical page number. Page locations in the L2 cache and in the memory are decoupled. Similar mechanisms have been previously used [82, 48].

the two modes can be selected for best performance.

The values for *TID* and *BIN* are assigned by the OS virtual memory management module at the time of a page fault. The OS can calculate these two fields by following simple heuristics, such as bin hopping and round robin techniques [52]. *TID* and *BIN* can also be determined using a sophisticated mechanism which may require additional hardware support. For the schemes proposed in this thesis, the main approach adopted is to calculate *TID* and *BIN* based on the data placement hints derived off-line. We assume these hints are generated by a compiler analysis routine and carried with a program binary. Strictly being “hints”, the data affinity information causes no harm if the OS and the hardware do not support any cache-level data affinity optimization. It simply falls back to the plain shared cache scheme. Given this new mechanism, a data block can reside virtually anywhere in a cache without regard to its original physical address. Thus the whole data block address has to be compared with cache tags in order to guarantee correctness.

The introduced extra fields incur a modest storage overhead, while offering a very flexible data placement mechanism. Assuming a 16-tile CMP with 256KB 8-way associative L2 cache slices, *TID* and *BIN* are 4-bit and 2-bit long respectively. This only results in less than 10% increase in page table size, if the address width is 64-bit. Since the page table resides in main memory, whose capacity is not a big concern in today's computer, the page table expansion is acceptable. The extra tag bits introduce less than 1% cache area overhead for a 64-byte cache block size. The *TID* and *BIN* values are assigned at the time when a page mapping is created. They persist until the corresponding page is replaced out of the main memory. Therefore there is no consistency issue. No TLB flush is required.

3.2 EXPERIMENTAL SETUP

To simplify and speed up the evaluation process, different simulation infrastructures are used for single-threaded programs and multithreaded programs respectively. For single-threaded programs, we extended the SimpleScalar tool set (v3.0) [8] to model a tile-based CMP architecture as described in Section 3.1. Since only one single-threaded program is simulated on this platform at a time, the modified simulator essentially models a tile-based cache structure. A core within a non-active tile is assumed to have no load and thus not modeled. The tile on which a benchmark program runs contains a core with a 4-issue out-of-order pipeline. The 32KB 2-cycle L1 instruction and data caches are 2-way set associative and the 256KB 8-cycle L2 cache slice is 4-way set associative. An L1 cache block is 64-byte wide while an L2 cache block is 128 bytes.[1] A miss in an L1 cache triggers a request sent through the on-chip network to the home L2 cache. Each hop in the networks takes 5 cycles. The main memory access latency is 300 cycles. Otherwise stated, a program always runs on tile 5. We selected 11 integer and 7 floating-point programs from the SPEC2k CPU benchmark suite [95]. Because the execution of a single-threaded program rarely involves coherence activities, the coherence mechanism is ignored for simplicity. After fast-forwarding

[1]This cache block size setting is common in a uniprocessor architecture [109].

Component	
Processor Model	in-order
Issue Width	2
L1 I/D Cache	
Cache Line Size	64 B
Cache Size / Associativity	8 KB / direct-mapped
Load-to-Use Latency	2 cycles
L2 Cache	
Cache Line Size	64 B
Cache Size / Associativity	128 KB / 8-way
Tag Latency	2 cycles
Data Latency	6 cycles
Replacement Policy	Random
Network on Chip	
Topology	4 x 4 2D mesh
Hop Latency	3 cycles
Main Memory Latency	300 cycles

Table 1: Baseline architecture configuration

through an initialization phase and having a warm-up period, statistics are collected during a period of 800M instructions.

In addition, we constructed a CMP cache system simulator by extending the timing interface within Simics [89] to evaluate our software-oriented shared cache management scheme for multithreaded programs. The simulated CMP adopts the UltraSPARC III ISA [97] and runs a version of the Solaris operating system [96]. It models a 16-tile CMP with a 4×4 2D mesh on-chip networks as shown in Figure 2. Each processing core has a two-issue in-order pipeline and private L1 instruction and data caches. Cache coherence is enforced by a distributed directory-based coherence protocol with MESI states [60]. Each router in the network has separate queues for messages incoming from different links. Router arbitrator fetches messages and delivers them to the next hop in a round-robin fashion. Table 1 describes the baseline architecture configuration. Because the benchmark programs we use have small working set size, the cache size is scaled in the experiments so that caches see reasonable pressure.[2] The extra level of page address translation for an L2 cache access as described in Figure 3 is mimicked in the simulator to avoid modifications in the OS kernel. The simulator maintains a translation table that approximates the contents of the actual page table in the OS. When there is an L2 cache access, the simulator looks up the translation table for the target tile ID and cache bin number based on the given virtual page number. On the first access of a page, data affinity hints are consulted to assign new values for TID and BIN. The data affinity hints are fed into the simulator directly at the beginning of a simulation. The overhead of making affinity decision in the OS is a small one-time cost and is ignored.

12 programs from the SPLASH-2 benchmark suite [86] and 2 programs from the PARSEC benchmark suite [76] are chosen as the latency-oriented multithreaded workloads for experiments. They are listed in Table 2 with associated inputs. There are a number of reasons for picking these benchmark programs. First, since the experiments involve page placement activities, a major part of which are done at the very initial stage of a program execution, it is necessary to simulate a program from the beginning to the end. That is, fast forwarding is not an option. Given the slow speed of a detailed CMP cache simulator, it is impossible to simulate a very large application. Second, as required by the experiments, function calls for dynamic memory allocation in a program's source code are manually replaced by wrapper functions to capture dynamic allocation information. This avoids implementing a full-blown compiler. This method works well for C programs, but has trouble with programs written in object-oriented languages. Third, the main focus of this work is on multithreaded programs. The SPLASH-2 and the PARSEC benchmark suites are the most commonly and widely used

[2]In fact, based on the study, we found that using a larger cache size has a limited impact on the relative performance of the evaluated schemes. Thus, the conclusions drawn from this study are also valid for larger cache configurations. This is because each individual thread in these benchmark programs has a small working set size that can mostly fit in a 128KB cache.

Program	*Small Input*	*Median Input*	*Large Input*
barnes	16K particles	32K particles	64K particles
cholesky	tk15.O	tk16.O	tk29.O
fft	256K data points	1M data points	4M data points
fmm	16K particles	32K particles	64K particles
lu	512 x 512 matrix (cont.)	1024 x 1024 (cont.)	2048 x 2048 (cont.)
ocean	258 x 258 grid (cont.)	514 x 514 grid (cont.)	1026 x 1026 grid (cont.)
radiosity	test	room	largeroom
radix	4M keys, 1024 radix	8M keys, 1024 radix	32M keys, 1024 radix
raytrace	teapot	car	balls4
volrend	scaleddown4	scaleddown2	head
water-ns	512 molecules	1000 molecules	2744 molecules
water-sp	512 molecules	1000 molecules	4096 molecules
blackscholes	64K options	128K options	256K options
swaption	4K swaptions	8K swaptions	16K swaptions
specjbb	1000 transactions	5000 transactions	10000 transactions
apache	100 requests	500 requests	1000 requests
btree	1000 requests	5000 requests	10000 requests
cheetah	1000 requests	5000 requests	10000 requests

Table 2: Benchmarks with small, median and large input parameters.

programs in the research community.

In addition, specjbb 2005 [95] and apache [4] are used as the throughput-oriented multi-threaded benchmarks. Specjbb evaluates the performance of server side Java by emulating a three-tier client/server system with emphasis on the middle tier. For the experiments, specjbb is set up with 16 threads which serve transactions for 16 warehouses simultaneously. apache is an open-source HTTP server program. The performance of apache is tested with its benchmarking tool *ab*, which is configured to establish 128 concurrent HTTP connections. We also include two customized kernels called btree and cheetah. Btree is a small database kernel, which simulates parallel search, insert, and delete operations with coarse-grain locking to guarantee data consistency. Cheetah [58, 92] is a light-weight HTTP daemon originally, modified to support multithreading. In order to fit the simulation environment, it is specially customized to be self-sustaining without the need of being driven by external requests. In this study, cheetah is configured to spawn 16 threads, serving random file fetch requests simultaneously.

Table 2 lists all multithreaded programs with their corresponding input sets used in our study. The different input sets are used for each program in our experiments in order to evaluate the generality of the off-line derived hints across different input sizes. A small input set is used to collect the trace. Then median and large input sets are used to report results. Throughput-oriented multithreaded programs do not require particular input data like latency-oriented multithreaded programs. Their execution length is solely controlled by the given number of transactions. In this study, we simulate throughput-oriented multithreaded programs for a small number of transactions for trace analysis. Then performance is evaluated with a much larger number of transactions.

For the experiments in this thesis, we conveniently used an architecture simulator to collect traces. However, in realistic software development settings, one would seek a faster tracing strategy. There exist much more efficient alternative solutions that use, for example, hardware performance counters [64] or binary instrumentation tools [66]. While building an optimized tracing tool can be rewarding, it is beyond the scope of the study in this thesis. Considering that programmers resort to various performance tuning tools to improve the performance of multithreaded programs, the extra one-time tracing overhead can be justified.

4.0 DATA PLACEMENT FOR SINGLE-THREADED PROGRAMS

In this chapter, the page placement strategies for single-threaded programs are introduced. Because a single-threaded program seldom has shared data, it is desirable to attract data as close to the running program as possible to reduce the access delay incurred by the on-chip networks. The problem becomes how to aggregate a program's working set in the L2 cache nearby the running program while keeping the miss rate low. In what follows, a *static 2D page coloring* technique is first discussed. It uses off-line analysis hints to achieve the near-optimal performance. This static scheme utilizes the profiling information of a whole program, which may not be practical in certain production systems. Motivated by the results of the static 2D page coloring scheme, a practical solution called *dynamic 2D page coloring* is then devised. The dynamic scheme tries to approach the performance of the static 2D page coloring scheme without performing the off-line profile. So it is a practical solution, which can be used for optimizing performance of programs in a real system. The results and conclusions are presented at the end.

4.1 STATIC 2D PAGE COLORING

In this section, the static data placement scheme for single-threaded programs is presented. It is called *static two-dimensional (2D) page coloring* because data to cache bin mappings are determined before a program's execution. A profile-driven method is employed to guide the mapping process. A cache bin is a smallest group of cache sets which would hold an entire memory page. The number of cache bins in a cache is simply the cache size divided by the product of the page size and the associativity. The proposed off-line algorithm greedily selects a desirable cache bin (among all cache slices) for a page based on the detailed inter-page conflict information derived from a program's memory reference trace.

This scheme is called "2D" page coloring because choosing a target cache bin decides not only the cache bin within a cache slice, but also the cache slice itself, which in turn determines the program-to-data distance. While collecting and analyzing detailed traces for all programs might be impractical, this static scheme provides valuable insights into the "ideal" program performance on distributed shared caches, by optimizing cache access latency and miss rate together. It is also noted that the performance of the proposed profile-driven 2D page coloring places an upper bound on the performance achievable by an aggressive static compiler analysis [20].

4.1.1 Basic Approach

The static 2D page coloring scheme consists of three phases: *trace generation*, *trace analysis* and *mapping hints generation*. In the trace generation phase, memory references of a target program are collected. To accurately capture the related cost in the trace analysis phase, only L2 cache references are collected. A sampling method can be used if the L2 cache references are too large to be handled efficiently. In the trace analysis phase, the number of references to different pages and the number of inter-page conflicts are counted, within the scope of the whole trace. While the number of references for each page can be easily obtained given a memory reference trace, computing the number of conflicts between pages is impossible before the mappings of these pages are known.

To tackle this problem, an assumption is made: if there are two references to page A and page B and there is no other reference to page B in between, these two references can potentially cause a conflict miss if page A and page B are placed in the same cache bin [82]. Based on this assumption, the algorithm for calculating conflict information is sketched in Figure 4. Two matrices **Reference**[ ][ ] and **Conflict**[ ][ ] are defined. The **Reference**[ ][ ] matrix keeps track of temporal relationships among references. The **Conflict**[ ][ ] matrix counts the number of potential conflicts between any pair of pages. To update the matrices, references in the trace are processed one by one, as shown in Figure 4. For each reference,

```
while trace is not empty {
    get the next reference R from trace
    PI = array index of the page accessed by R

    for(i = 0; i < total number of pages; i++) {
        Reference[i][PI] = 1;
        if(Reference[PI][i] == 1) {
            Conflict[PI][i]++;
            Reference[PI][i] = 0;
        }
    }
}
```

Figure 4: The algorithm to extract conflict information from an L2 cache access trace.

the column bits in **Reference**[ ][ ] corresponding to the accessed page are set to 1. All the bits in the row corresponding to the same page are then checked. Any bit previously set to 1 indicates that the current reference may cause a conflict with this previous reference. Thus, the corresponding position in **Conflict**[ ][ ] is increased by 1. After all references in a trace are processed, an item in **Conflict**[ ][ ] matrix, **Conflict**[**i**][**j**], contains the number of potential conflicts when page **i** and **j** are mapped to the same cache bin.

Figure 5 demonstrates this process using a simple example. In this example, page A, B, C, and D are accessed in the given order. Initially, all entries in both matrices are 0. After page A is accessed, entries from the column A of the **Reference**[ ][ ] matrix are marked to 1. In addition, entries from row A of the **Reference**[ ][ ] matrix are scanned. If a marked entry is found, the corresponding entry from the **Conflict**[ ][ ] matrix is increased by 1. In this case, nothing happens since all entries are 0. Access to page B repeats the same process: Entries from the column B of the **Reference**[ ][ ] matrix are marked to 1 and row B checked.

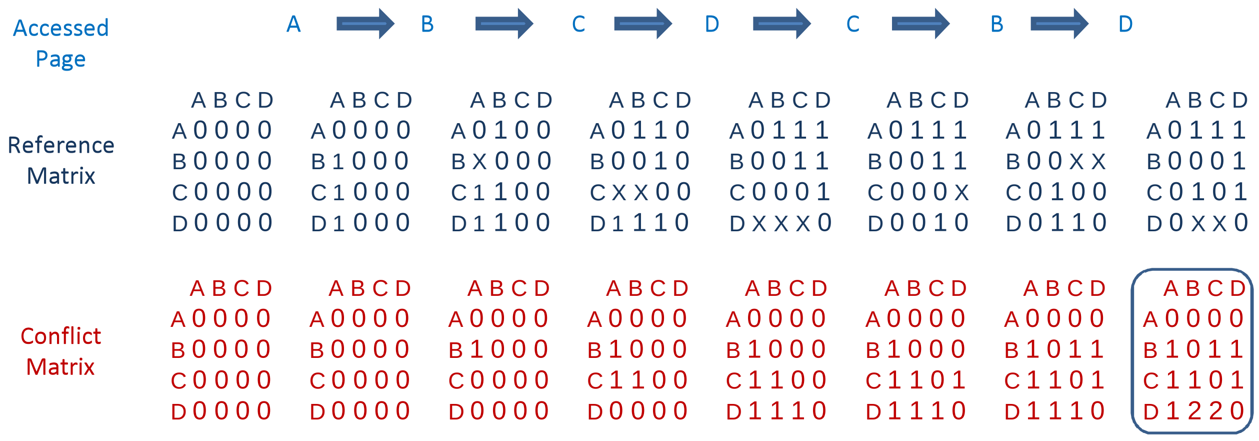

Accessed Page
A
B
C
D
C
B
D
Reference Matrix
A B C D / A 0 0 0 0 / B 0 0 0 0 / C 0 0 0 0 / D 0 0 0 0
A B C D / A 0 0 0 0 / B 1 0 0 0 / C 1 0 0 0 / D 1 0 0 0
A B C D / A 0 1 0 0 / B X 0 0 0 / C 1 1 0 0 / D 1 1 0 0
A B C D / A 0 1 1 0 / B 0 0 1 0 / C X X 0 0 / D 1 1 1 0
A B C D / A 0 1 1 1 / B 0 0 1 1 / C 0 0 0 1 / D X X X 0
A B C D / A 0 1 1 1 / B 0 0 1 1 / C 0 0 0 X / D 0 0 1 0
A B C D / A 0 1 1 1 / B 0 0 X X / C 0 1 0 0 / D 0 1 1 0
A B C D / A 0 1 1 1 / B 0 0 0 1 / C 0 1 0 1 / D 0 X X 0
Conflict Matrix
A B C D / A 0 0 0 0 / B 0 0 0 0 / C 0 0 0 0 / D 0 0 0 0
A B C D / A 0 0 0 0 / B 0 0 0 0 / C 0 0 0 0 / D 0 0 0 0
A B C D / A 0 0 0 0 / B 1 0 0 0 / C 0 0 0 0 / D 0 0 0 0
A B C D / A 0 0 0 0 / B 1 0 0 0 / C 1 1 0 0 / D 0 0 0 0
A B C D / A 0 0 0 0 / B 1 0 0 0 / C 1 1 0 0 / D 1 1 1 0
A B C D / A 0 0 0 0 / B 1 0 0 0 / C 1 1 0 1 / D 1 1 1 0
A B C D / A 0 0 0 0 / B 1 0 1 1 / C 1 1 0 1 / D 1 1 1 0
A B C D / A 0 0 0 0 / B 1 0 1 1 / C 1 1 0 1 / D 1 2 2 0

Figure 5: An example demonstrates how the algorithm extracts conflict information from an access trace. An X in **Reference**[ ][ ] matrix indicates a detected conflict. The **Conflict**[ ][ ] matrix in the rounded box contains the derived conflict information for the processed trace.

Since the entry **Reference**[**B**][**A**] is 1, a potential conflict is detected (represented by X in the matrix). The **Reference**[**B**][**A**] is added by 1. The final conflict information is provided by the **Conflict**[ ][ ] matrix. For instance, if page D is assigned to a cache bin where page B is allocated, it could raise 2 additional conflict misses. Interestingly, if page B is mapped to a cache bin where page D resides, it only causes 1 potential conflict miss.

Note that the numbers in **Conflict**[ ][ ] have a lot of false conflicts because the conflicts are estimated at the page granularity. Even though two pages are mapped to the same cache bin, they may not cause conflicts if different portions from these two pages are actually accessed. In addition, an implicit assumption in this algorithm is that the target cache is direct-mapped. If a cache has set associativity, a number of pages up to the associativity can be placed in a set without a conflict.

The inter-page conflict information is used in the last phase when estimating the extra potential cache misses caused by placing a page to a cache bin. The goal in this phase is to minimize the total cost of all L2 cache accesses. This is achieved by iteratively computing the cost of assigning a particular page to all cache bins and selecting the cache bin with the smallest cost. Given the inter-page conflict information in **Conflict**[ ][ ] and other necessary microarchitectural parameters, hints for mapping pages to cache bins can be calculated. Since the page coloring problem is in general NP-complete [52], a heuristic method is adopted to make the computation tractable. The proposed coloring algorithm evaluates pages from the one with the largest number of accesses and proceeds in a decreasing order. The cost of assigning a particular color or bin **C** to a page **P** is computed by the following cost function:

$$\begin{aligned}\mathrm{Cost(P,\ C)} &= \alpha \times \mathrm{TotalConflicts(P,\ C)} \times \mathrm{MemLatency} \\ &\quad + (1-\alpha) \times \mathrm{TotalAccesses(P)} \\ &\quad \times (\mathrm{L2Latency} + \mathrm{NoCDelay(C)}) \qquad (4.1)\end{aligned}$$

In Equation 4.1, **TotalConflicts(P, C)** is given as $\sum$ **Conflict**[**P**][**j**]/**N** for any page **j** already mapped to **C**. **N** stands for the number of pages that have been allocated to the cache bin. **NoCDelay(C)** denotes the transmission latency on the on-chip networks. Without losing generality, it is assumed in Equation 4.1 that the program location is fixed (and thus not shown) for the clarity of presentation.

Since **TotalConflicts(P, C)** is an estimation of the page conflicts, it is not proportional to **TotalAccesses(P)** due to the false conflict issue mentioned previously. So a parameter α is introduced to balance the weights between **TotalConflicts(P, C)** and **TotalAccesses(P)**. α can have a value ranging from 0 to 1. It essentially controls the *page aggregation density*. With a smaller α, more weight is put on **NoCDelay()**, thus placing pages closer to the program location. As such, when α is 0, the algorithm simulates a private cache. On the other hand, with α equal to 1, the algorithm simulates a shared cache by only considering the aggregated miss penalty and ignoring the on-chip network delay and cache access latency. The process of selecting an optimal cache bin for a page using the above cost function is repeated until all pages are colored. The derived color assignment information is then used to guide the page placement at run time [27].

4.1.2 Discussions

Although the focus of the proposed scheme is on the performance of a single-threaded program, it is straightforward to extend the presented algorithm to handle co-scheduled programs. For instance, it requires little change to Equation 4.1 to accommodate a set of co-scheduled programs that do not have data sharing (*i.e.*, multiprogrammed workload). For a multithreaded workload, however, the access cost component (the second term) needs to be modified to include the impact caused by the sharing behavior among threads. Equation 4.2 shows the new cost function.

$$\begin{aligned}
\text{Cost(P,C)} \quad = \quad & \alpha \times \text{TotalConflicts(P,C)} \times \text{MemLatency} \\
& + (1-\alpha) \times (\textstyle\sum \text{TotalAccesses(P,S)} \\
& \times (\text{L2Latency} + \text{NoCDelay(C,S)})) \\
& \text{for all sharers S.}
\end{aligned} \tag{4.2}$$

The new cost function takes into account the number of accesses from different sharers and the network delay based on the locations of all sharers. This extension may not always lead to maximum performance for a multithreaded workload because the changing sharing behavior is not captured and exploited by this once-and-for-all mapping strategy. In addition, a popular page shared by several threads may end up in the center relative to all sharers, not benefiting anyone. A dynamic scheme could prove more effective in this case.

4.2 DYNAMIC 2D PAGE COLORING

4.2.1 Basic Approach

In this section, a dynamic 2D page coloring scheme is presented. It optimizes both cache hit latency and cache miss rate. The dynamic page placement exploits the on-line cache resource usage information to select a home cache bin having the smallest total expected cache access cost for a new page. The key design issues for the proposed scheme are: (1)

what information to collect and (2) how to compute the expected cache access cost with the information.

What run-time information do we collect? Selecting a good home cache bin for a page involves estimating the combined cost of cache access latency and cache misses. The cache access latency is simply the distance between the program and the target cache slice. The process of estimating the number of cache misses, however, requires it to continuously monitor the *cache bin hotness*. Cache bin hotness measures the level of conflicts a cache bin experiences in a given interval. Because accurately predicting future page access behavior is difficult, the initial placement decisions are based on the most recent hotness information of the cache bins in consideration. There are potentially many different ways to assess the hotness of a cache bin. For example, one can measure the cache pressure by counting the number of hot pages [27] or derive the cache utility by tracking the LRU stack positions being touched [78]. In this study, we examine an alternative method which simply counts the number of misses (**BinMiss()**) and accesses (**BinAcc()**) at each cache bin because the values are directly used in the cost estimation process.

How do we determine a home tile for a page? The actual cost of placing a new page to a cache bin **C** is computed at run time using Equation 4.3. The estimated cost in Equation 4.3 is simply the average L2 cache access latency to the bin **C**. Because there is no information about the page usage at the time of the initial page placement, the cost function only considers the current state of a cache bin. The experimental results indicate that page placement decisions guided by the proposed cost function are effective. In order to reflect the phase changes of a program, **BinMiss()** and **BinAcc()** need to be continuously decayed. The decay period ($\mathbf{T_{decay}}$) is a parameter to the overall cache management scheme. To determine a home tile for a page, it simply chooses a cache bin having a minimum cost.

$$\begin{aligned}\mathrm{Cost(C)} &= \frac{\mathrm{BinMiss(C)}}{\mathrm{BinAcc(C)}} \times \mathrm{MemLatency} \\ &\quad + (\mathrm{L2Latency} + \mathrm{NoCDelay(C)})\end{aligned} \tag{4.3}$$

4.2.2 Page Migration

While efforts are made to pick the best target cache bin in the initial page placement step, a program's changing memory access behavior and the dynamicity created by the OS process scheduling may result in largely uneven usage of cache resources. To secure the program performance under such circumstance, the dynamic 2D page coloring scheme can employ a dynamic page migration strategy. Page migration helps improve performance in two complementary ways. First, it balances cache bin usages and reduces miss rate by moving a page from a hot bin to a cold bin. Second, it can improve cache access latency by relocating a hot page from a remote cache slice to a cache slice closer to the program.

In this page migration scheme, it identifies a page having a large expected migration benefit, and migrate cache blocks belonging to the page to a new cache bin. This process is performed based on the run-time cache usage information. Determining where to migrate a page is similar to finding a best cache bin at the initial placement step. The guiding principle is to consider both cache access latency and miss rate when evaluating the expected cache access cost. While the design space for the page migration scheme is wide, here it uses a cost-driven approach to identify a page for migration, similar to how it decides the home tile for a newly allocated page. The key design issues are: (1) identifying a page to migrate, (2) deciding when to migrate a page, and (3) controlling the frequency of migration actions, given the frequency of workload changes and the overhead of migration. These issues are inter-related.

The cost of a cache access incurred on a page **P** is evaluated on every L2 cache access. Since the access behavior for a newly allocated page may not represent the condition of its stable usage, the initial L2 cache accesses do not trigger the evaluation. Only when the **PageAcc(p)** exceeds a threshold value $\mathbf{Th_{eval}}$, the monitoring process is enabled. A migration is triggered for a high-cost page if there exists a cache bin which will render a better home for the page as shown in Equation 4.4.

$$(\mathrm{Cost}(\mathrm{Bin}_{\mathrm{cur}}) - \mathrm{Min}(\mathrm{Cost}(\mathrm{Bin}_{\mathrm{any}}))) > \mathrm{Th}_{\mathrm{mig}}, \tag{4.4}$$

The $\mathrm{Th}_{\mathrm{mig}}$ in Equation 4.4 is a threshold value, which controls the frequency of migration events. If $\mathrm{Th}_{\mathrm{mig}}$ is given a small value, page migration becomes very aggressive. The usage variation of a cache bin can cause its access cost becomes temporally lower or higher than the cache bin where the page currently resides. The page ends up thrashing between two cache bins with similar cost. Given the large overhead of a page migration, it can severely harm a program's performance. On the other side, a large threshold value makes the page migration to react to environment changes too slowly, losing optimization opportunities. Thus choosing a proper $\mathbf{Th}_{\mathbf{mig}}$ is important. In this study, we set $\mathbf{Th}_{\mathbf{mig}}$ to be one standard deviation of all cache bin costs. The cost function in Equation 4.4 for calculating the cost of placing a page **P** in a cache bin **C** is given in Equation 4.5. Naturally, the migration scheme chooses a cache bin with a minimum cost as the migration target.

$$\begin{aligned}\mathrm{Cost(C)} &= \mathrm{PageMiss(P)} \times \frac{\mathrm{BinMiss(C)}}{\mathrm{BinMiss(Bin_{cur})}} \times \mathrm{MemLat.} \\ &\quad +\mathrm{PageAcc(P)} \times (\mathrm{L2Lat.} + \mathrm{NoCDelay(C)})\,. \end{aligned} \tag{4.5}$$

4.2.3 Architectural Support

Figure 6 shows a block diagram of the microarchitectural support needed for the proposed dynamic 2D page coloring scheme. There are several structures added to facilitate dynamic 2D page coloring. First, each cache bin in the L2 cache is augmented with a pair of counters, **BinMiss()** and **BinAcc()**. They are updated on every L2 cache access. These counters can be implemented using registers to provide fast access. For a cache with a large associativity, the number of cache bins is usually small. So the hardware cost is negligible. Moreover, the counters are updated in parallel with an L2 cache access. Thus, they do not incur any delay.

Second, each tile maintains a **Global Bin Usage Table**, which provides information about how cache bins are used across the chip. Each entry in this table provides the cache usage information of the optimal cache bin from one tile. The usage information includes **BinIndex**, **BinMiss()** and **BinAcc()**. Other sub-optimal cache bins are not considered during the page placement or migration process. So it is not necessary to keep their information. As a result, the size of the table is proportional to the number of cores in a CMP. For a

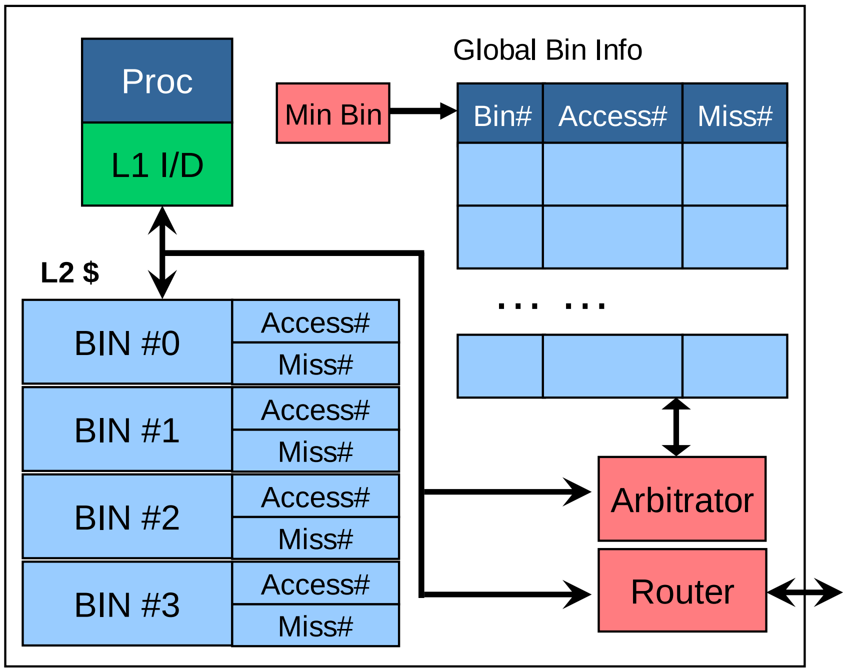


Figure 6: Microarchitectural support for dynamic 2D page coloring.

16-core CMP, the hardware overhead is very limited as will be shown below. It is noted that the information carried in this table is only a hint for making the page placement or migration decision; inconsistent and imprecise information is acceptable. Henceforth, the global bin usage table can be updated from time to time, via periodic messages between tiles, or even utilizing piggybacked information in regular messages in order to reduce synchronization and communication overhead.

When looking for the best cache bin for a page placement, calculations outlined in Equation 4.3 have to be performed efficiently. A straightforward solution is to use a hierarchy of comparators. At the bottom level, every pair of cache bins from the **Global Bin Usage Table** are compared in parallel. Then at the next higher level, the better one of one pair is again compared to the better one from another pair. This process continues until it reaches the top of the hierarchy, where the best cache bin is found. At each level, the number of cache bins to compare is halved. To find the best cache bin from 16 tiles, 4 levels of comparators are required. As the number of tiles increases, the number of levels in the comparator hierarchy grows as well. The hardware cost can become substantial, not to mention the increased delay. Instead of determining the best cache bin at the time of a page placement, an improved solution is to track the best cache bin continuously so that comparisons are done earlier. A pointer called **MinBin** is used to indicate the current best cache bin in the **Global Bin Usage Table**. Whenever there is an update in **BinMiss()** and **BinAcc()** in the table, the cost of the updated cache bin is calculated and compared to the cache bin. **MinBin** is redirected to the new cache bin if its new cost is lower. With the new scheme, only one comparator is needed. The optimal cache bin is given in **MinBin** and ready for use immediately.

The hardware cost for tracking cache bin usage information is modest. Assuming a counter width of 16 bits, the cost for each bin is only 4 bytes. Assuming that the page size is 8KB and a 256KB 8-way associativity L2 cache slice, only 16B storage is needed for the attached counters for each tile. The cost of **Global Bin Usage Table** is also negligible. Given the same cache configuration, a 16-core CMP contains 64 cache bins in total. Each bin requires Two counters and a bin index in **Global Bin Usage Table**, which only cost 5 bytes at most. Thus the total cost for one **Global Bin Usage Table** is estimated to be 320 bytes, only accounting for about 0.1% of total cache area.

4.3 EVALUATION RESULTS

For the sake of consistency, results reported are relative to the baseline shared cache scheme ("**SharedBase**"), which adopts simple heuristics to color pages [52]. No profile information is used for the baseline shared cache scheme.

4.3.1 Static 2D Page Coloring

Figure 7 shows how a change in α (in Equation 4.1) affects the page mappings and cache access behavior. 9 different values of α, ranging from 0 to 1, are examined. The α value increases at a stride of $\frac{1}{8}$. When α equals to 0, the factor of cache contention is not considered. So the static 2D page coloring scheme essentially simulates a private cache. On the other

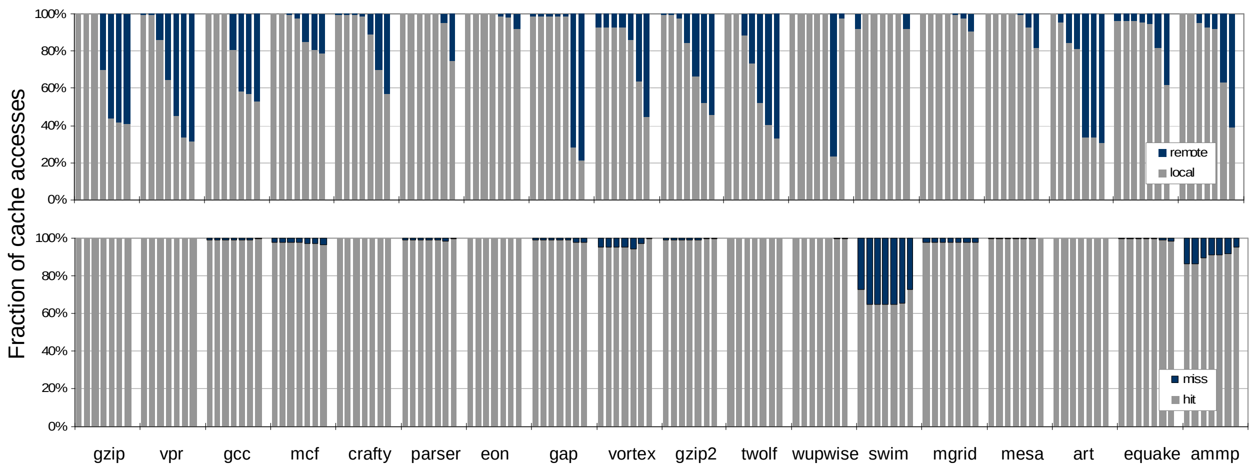


Figure 7: L2 cache access decomposition: local vs. remote (upper) and hit vs. miss (bottom) at different α values. Nine bars for each program represent cases for $\alpha = 0$, $\alpha = 1/8$, ..., $\alpha = 1$.

extreme, when α is set to 1, it behaves like a conventional shared cache without regard to the access delay on the networks.

The upper graph shows the distribution of accesses between a local cache slice or other remote cache slices. It clearly shows that increasing the α value scatters more cache accesses to remote cache slices, since the static 2D coloring scheme behaves more like a shared cache with large α value. The bottom graph shows the distribution of accesses between cache hit and cache miss. Programs like *gzip*, *twolf* and *art* have a very small number of misses, which does not change much when the α value varies. Many of their conflict misses are removed by the static 2D page coloring scheme ("**Static2D**" from now on) and unavoidable cold misses remain. *crafty* and *eon* have similar characteristics. As a result, they achieve their best performance with a small α value among those examined. Other programs like *mcf*, *parser*, *vortex*, *bzip2* and *swim* all exhibit a concave curve, which peaks roughly at the middle. *mgrid* and *equake* even show an increase in the number of misses. This situation is caused by imperfect trace information used in the off-line analysis algorithm. The heuristic

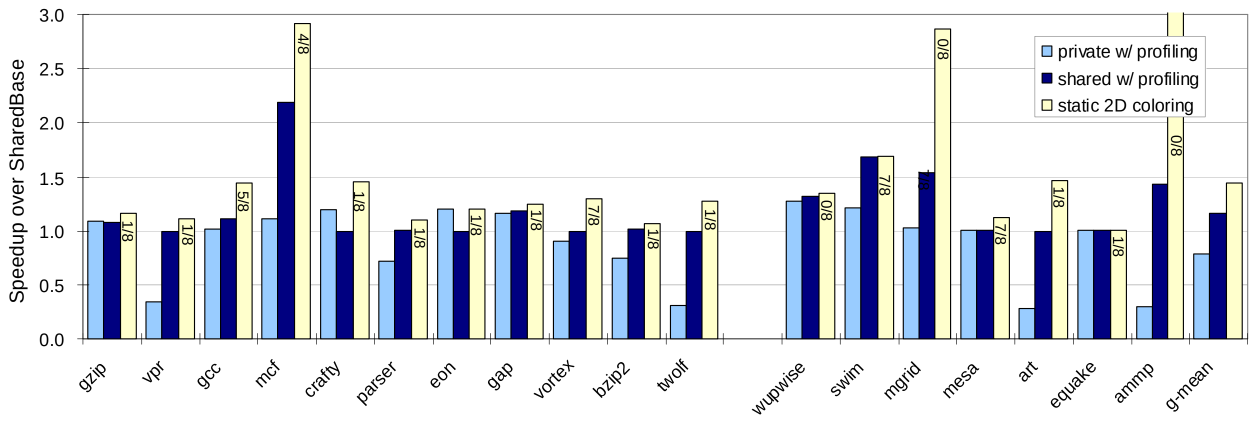


Figure 8: Performance of **Private**, **Shared**, and **Static2D**. Numbers on bars show the value of α chosen.

coloring algorithm cannot always produce the optimal page placement hints. When it fails to capture a program's cache access behavior, the number of cache misses increases from the optimal. In addition, when α equals to 0, no conflict information is considered. This may lead to non-deterministic behavior as exhibited in some programs such as the elevated miss rate in *parser*, *vortex* and *swim*. In general, a value of α which balances miss rate and latency yields the best performance.

Figure 8 shows the performance of the private cache scheme, the shared cache scheme and the proposed **Static2D**. For fair comparison purpose, an aggressive profile-guided page coloring technique [82] is applied on the private cache scheme and the shared cache scheme. They are named "**Private**" and "**Shared**" respectively to differentiate from **SharedBase**.

Note that all performance numbers are normalized to that of **SharedBase**. It is shown that **Static2D** consistently outperforms **Private** and **Shared**. The performance of **Private** often suffers due to the relatively small cache slice size of 256KB; *vpr*, *twolf*, *art*, and *ammp* are among the most affected. It exhibits a high L2 cache miss rate in these programs, which cannot be simply compensated by L2 cache latency savings.

Shared always shows better performance than **SharedBase** by reducing the number

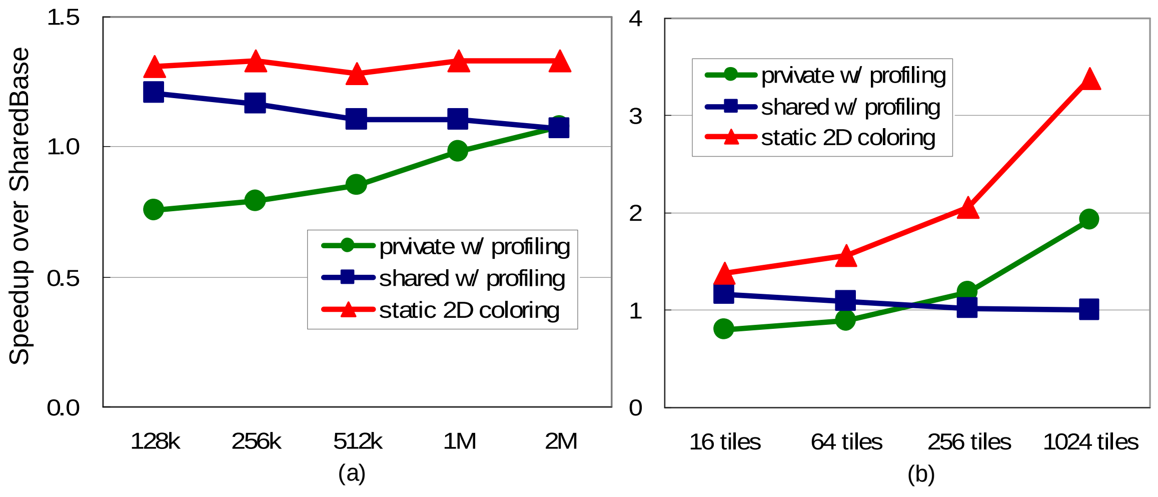


Figure 9: Average performance when (a) cache slice size is varied and (b) tile count is varied.

of conflict misses. The average access latency is not affected since no data is aggregated. Programs like *mcf*, *swim*, *mgrid* benefit much from the miss rate reduction and achieve over 50% performance improvement. **Static2D** achieves higher performance than both **Private** and **Shared** by balancing cache miss rate and cache access latency. *swim* is a notable exception, for which **Shared** achieves a better miss rate than **Static2D** due to its fine-grained block interleaving. On average, **Static2D** achieves 44.7% performance improvement over **SharedBase**, 23.7% over **Shared**, and 83.2% over **Private**.

Figure 9 shows how performance of different schemes scales when the cache slice size or the tile count changes. When cache slices are small, miss rate is the dominant performance factor. In this case, **Private** performs poorly. As the cache slice size increases, however, the gap between **Private** and **Shared** decreases. **Private** begins to outperform **Shared** at 2MB, where the performance loss caused by conflict misses is mitigated by large cache capacity and the cache access latency becomes a determining factor. Though not plotted, **Private** and **Static2D** will merge finally and **Shared** will approach 1 (*i.e.*, degenerate to **SharedBase**) as the cache size increases indefinitely. When there are more tiles on a chip, the average cache access latency of **Shared** and **SharedBase** grows (also shown in Figure 9(b)). **Shared** is shown to approach 1. That is, **Shared** degrades to the **SharedBase** on a large-scale CMP.

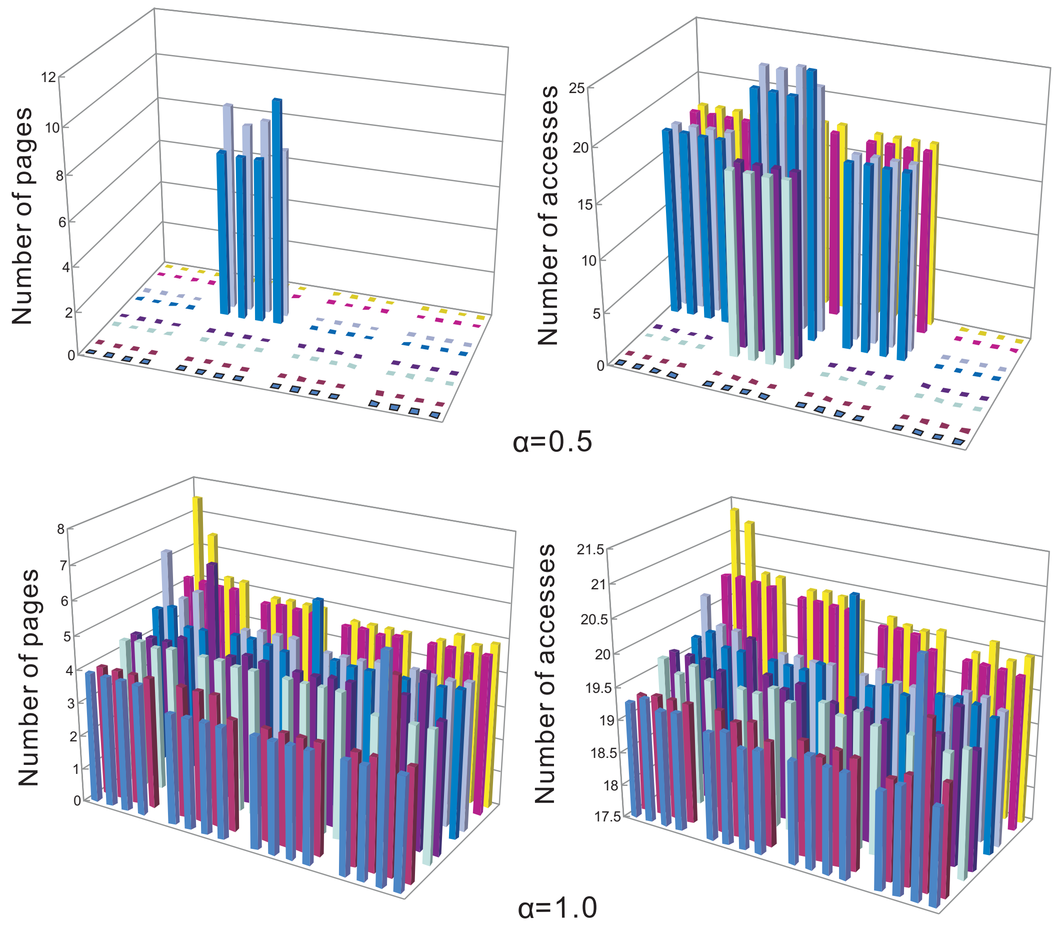


Figure 10: Page placement and cache access distribution for *ammp* with $\alpha = 0.5$ and $\alpha = 1.0$. Each bar in x-y space corresponds to a cache bin. A cluster of eight bars represents a cache slice from one tile.

This is because the latency becomes the dominant factor and the benefit of profile-guided coloring becomes negligible. By comparison, the performance of **Private** and **Static2D** is largely insensitive to the addition of new tiles. **Private** begins to outperform **Shared** due to the remote access latency saving on large-scale CMPs.

Lastly, Figure 10 gives an example of how **Static2D** allocates pages to different L2 cache bins. In case of ammp, if α is set to 0.5, most pages are allocated to the local cache slice while

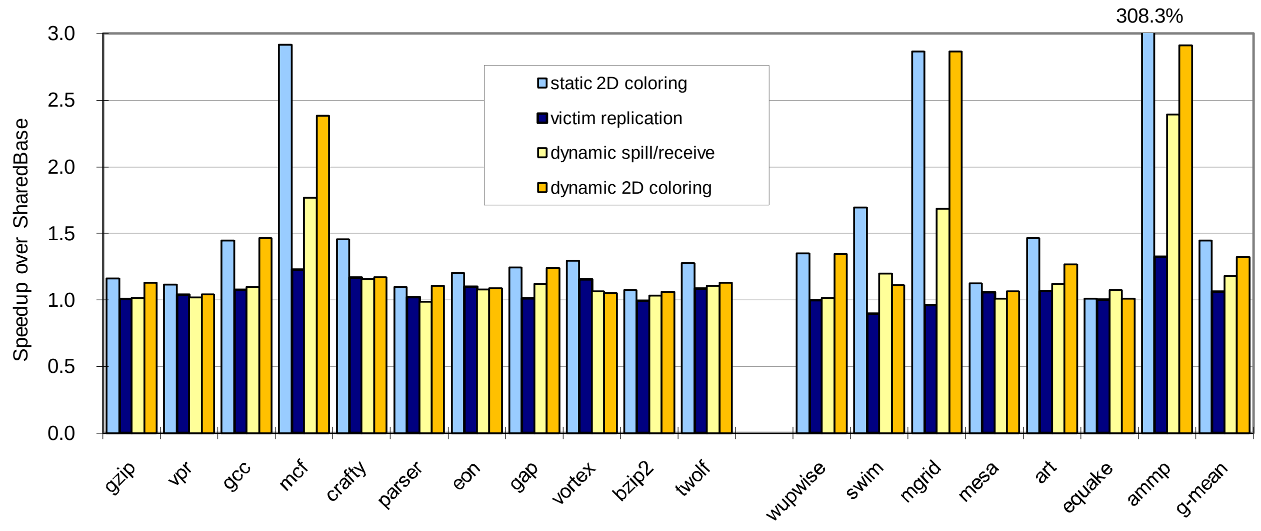


Figure 11: Program performance of the dynamic 2D page coloring scheme.

a few pages are spread out to neighboring tiles. The access distribution exhibits a similar pattern. After increasing α to 1.0, **Static2D** simulates **Shared** with data distribution at page granularity. Pages and memory accesses are almost evenly distributed across the whole CMP. It is clearly shown that a larger α value results in more spread memory accesses among the cache bins.

4.3.2 Dynamic 2D Page Coloring

For dynamic 2D page coloring scheme, $\mathbf{T_{decay}}$ is an important parameter, which controls the sensitivity to the program phase change. The results presented in this section are evaluated with a empirically chosen value (8,192 L2 cache misses).

Figure 11 shows the performance of **Static2D**, the victim replication scheme [111] ("**VR**"), the dynamic spill and receive scheme [79] ("**DSR**") and the dynamic 2D page coloring scheme without page migration ("**Dyn2D**"). Again, the results are normalized to the performance of **SharedBase**. **VR** achieves better performance than **SharedBase** for most of the studied programs. For *swim* and *mgrid*, **VR** degrades performance considerably. This degradation is mainly caused by the interference introduced by replications. *swim* and *mgrid* have large footprint. However, some of the accessed data have poor locality. Replication of these streaming data does not help but only pollute the cache content, kicking out other frequently reused data. Overall, the performance improvement of **VR** over **SharedBase** is shown to be limited. This is not surprising for single-threaded programs. Since only one tile is active during the program execution, the victim replication scheme essentially tries to duplicate the whole working set in the local cache slice. Commonly, the capacity of one cache slice could not satisfy the demand of a program in such conditions. **DSR** outperforms **VR** considerably by spreading the evicted data across all available cache slices. However, its performance is inferior to **Static2D** since the latter tries to utilize nearby capacities while **DSR** spreads data randomly. Another reason is that **Static2D** also considers the cache conflict condition when placing the data while **DSR** does not utilize this information.

Dyn2D provides slightly lower performance than **Static2D** overall. This is reasonable since **Dyn2D** makes a page placement decision solely based on the current cache usage while **Static2D** utilizes cache access information for the whole execution. The lack of future access information in **Dyn2D** can sometimes lead to poor decisions. For *mcf* and *swim*, the dynamic scheme performs considerably worse than **Static2D**. This is because these two programs access a large number of pages for a relatively small number of times per page. As a result, the cache usage changes rapidly. The dynamic scheme cannot react to the page usage changes in a timely manner.

Static2D and **Dyn2D** perform significantly better than **VR**. The improvement comes from two factors: (1) The page coloring schemes place data close to the program location when it is not able to fit data in the local cache slice. **VR**, on the other hand, does not provide such flexibility; and (2) The page coloring schemes provide extra benefit by minimizing miss rate through cautious data placement. In contrast, **VR** can potentially introduce more misses because of the increased local cache pressure by injecting replicas without control. Overall, the performance of **Dyn2D** is comparable to that of **Static2D**. They boost the performance by 32.3% and 40.9%, respectively, compared with **SharedBase**. Compared to **VR**, **Dyn2D** gains 24.7%.

Figure 12 takes *mgrid* as an example to show how different schemes, **SharedBase**, **Shared**, **Static2D**, and **Dyn2D** (from (a) to (d)), create changes in how frequently cache

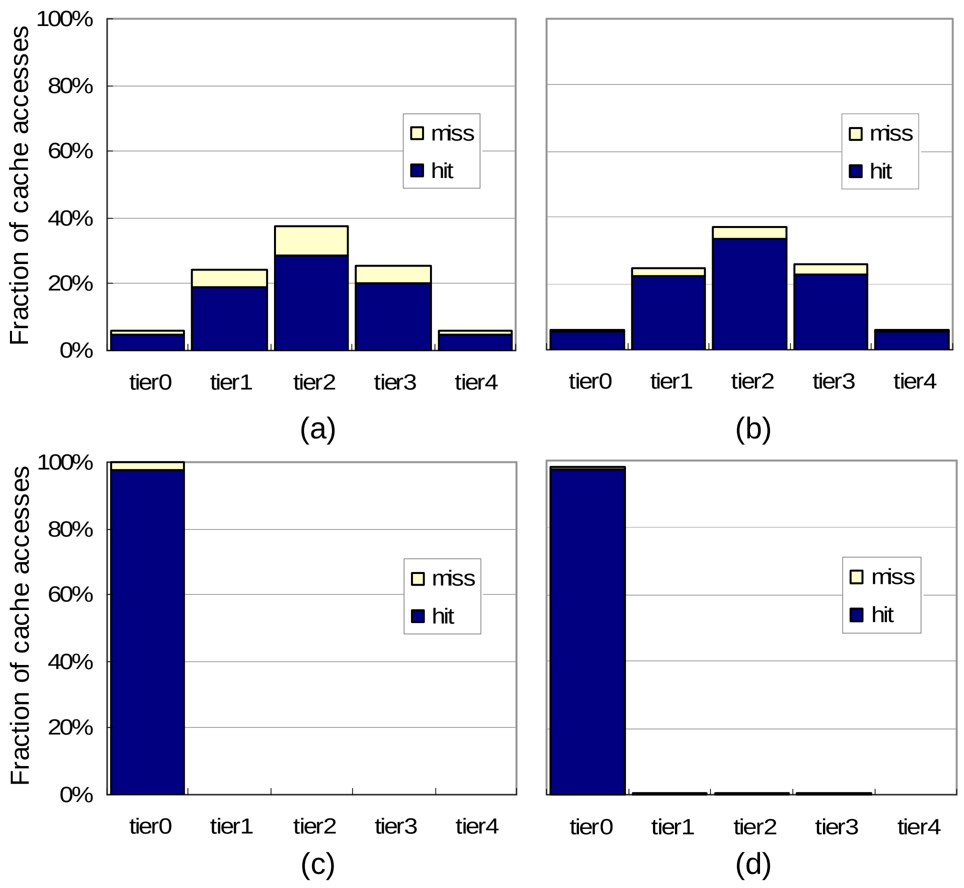


Figure 12: Cache access distribution of *mgrid* under 4 different schemes: (a) **SharedBase**, (b) **Shared**, (c) **Static2D**, and (d) **Dyn2D**.

slices (grouped into *tiers*) are accessed and how often accesses hit. Tier 0 refers to the local cache slice, tier 1 refers to the four neighbors in north, south, west, and east, and so on. Compared with **SharedBase**, **Shared** does not change the access frequencies to different tiers. However, it reduces the miss rate from over 22% down to 10.4%, resulting in an 1.5× speed-up. **Static2D** further reduces the miss rate to 2.2%, while attracting almost all pages to the local cache slice. Lastly, **Dyn2D** trades access latency for an even lower miss rate, thus achieving the best performance.

Figure 13 compares the performance of the dynamic 2D page coloring scheme without page migration and with page migration enabled. For those memory intensive workloads,

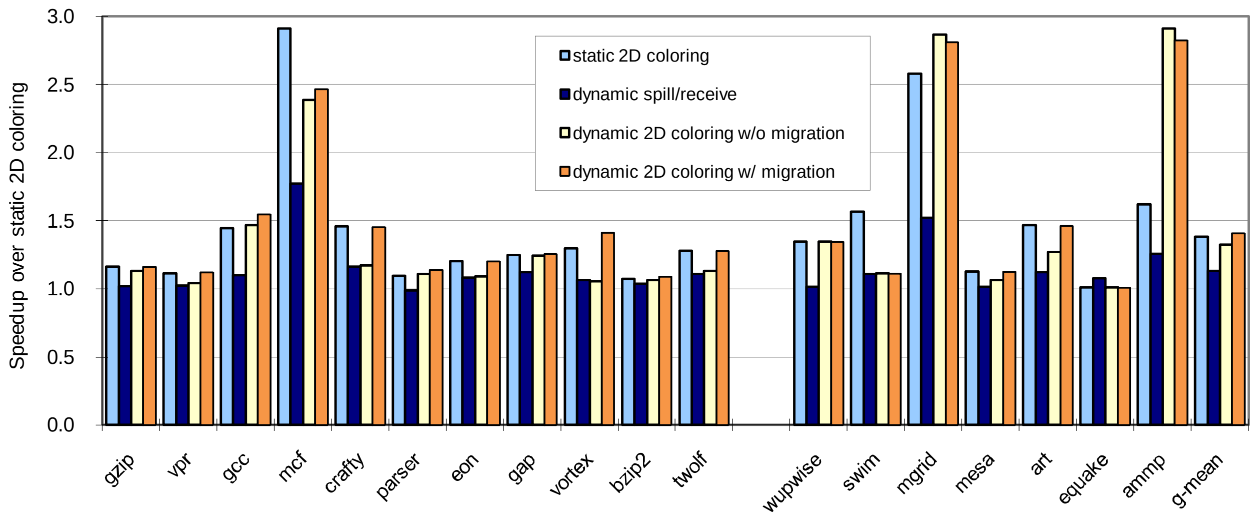


Figure 13: Program performance of the dynamic 2D page coloring scheme with and without page migration enabled normalized to the performance of static 2D page coloring scheme.

such as *twolf*, *vortex*, and *art*, the page migration scheme achieves significant performance improvement. The performance of *vortex* is improved by nearly 34.2%, compared to the plain dynamic 2D page coloring scheme. Memory intensive programs are sensitive to L2 cache access latency and conflict miss changes. Other programs only see moderate increase in performance. Two programs, *mgrid* and *ammp*, suffer about 3% performance loss when page migration is enabled. These two programs have large working set. The cache contention cannot be mitigated by page migrations. As a result, the false triggering of page migrations incur unnecessary overhead but no benefit. On average, the performance improvement over the plain dynamic 2D page coloring scheme is 6.9%.

4.3.3 Multiprogrammed Workload

The multiprogrammed workload is a generalized variation of the single-threaded workload as no system runs only one single-threaded program at a time. So the dynamic 2D page coloring scheme is also evaluated for multiprogrammed workloads. This study is conducted on a full-system simulator built on Simics [89]. The simulator models the same machine configuration used in experiments for single-threaded programs. To form workloads, programs are grouped into three classes: low pressure, medium pressure, and high pressure, based on their cache usage. A combination of eight programs are selected for each workload class. In the first workload labeled "mix.low," the programs include *gzip*, *crafty*, *parser*, *eon*, *vortex*, *vpr*, *mesa*, and *swim*. The workload is designed to mimic a situation where each co-scheduled program requires a small L2 cache space. The next workload, labeled "mix.mid," has *wupwise*, *gzip*, *mcf*, *crafty*, *parser*, *mgrid*, *eon*, and *art*. Lastly, the workload "mix.high" comprises of *twolf*, *gap*, *mcf*, *wupwise*, *mgrid*, *art*, *equake*, and *ammp*.

After skipping the initialization phase of all programs, the progress of every program in the group is measured after one simulated second. Program locations are fixed, ranging from tile 4 to tile 11. Additionally, the decay method in the dynamic 2D coloring scheme is modified slightly as follows. Counters are right-shifted on every 50k cycles. Then the values in the cache bin miss counters are added to the cache bin access counters in order to better distinguish between hot bins and relatively cold bins that have many cold misses. The reason for doing this is that after every decay the newly generated misses get more weight on the calculated miss rate. So miss rate can vary substantially, causing unwanted results. Adding the miss count to the access count decreases the miss rate and hence the effect of the newly generated misses after a decay. There may be many different implementations to achieve the same effect, but this simple method works sufficiently well in the experiments.

Figure 14 shows the performance of the three workloads in terms of averaged speedup (a) and aggregate throughput (b). As expected, the performance of the private cache scheme keeps decreasing (compared with that of the shared cache scheme) as the L2 cache demand of the workloads increases. When the L2 cache contention is high ("mix.high"), the private cache scheme performs poorly, worse than **SharedBase**. **VR** follows the similar trend as local contention degrades the performance of data replication considerably. **DSR** intelligently avoids local contention by spilling the data to spare cache capacity, thus achieving better performance than private cache scheme and **VR**. The **Dyn2D** outperforms both the private and the shared cache scheme robustly, in almost all the comparison points. In particular, it achieves an improvement of up to 26.4% using the average speedup metric and 38.6% using the aggregate throughput metric when compared to the baseline shared cache

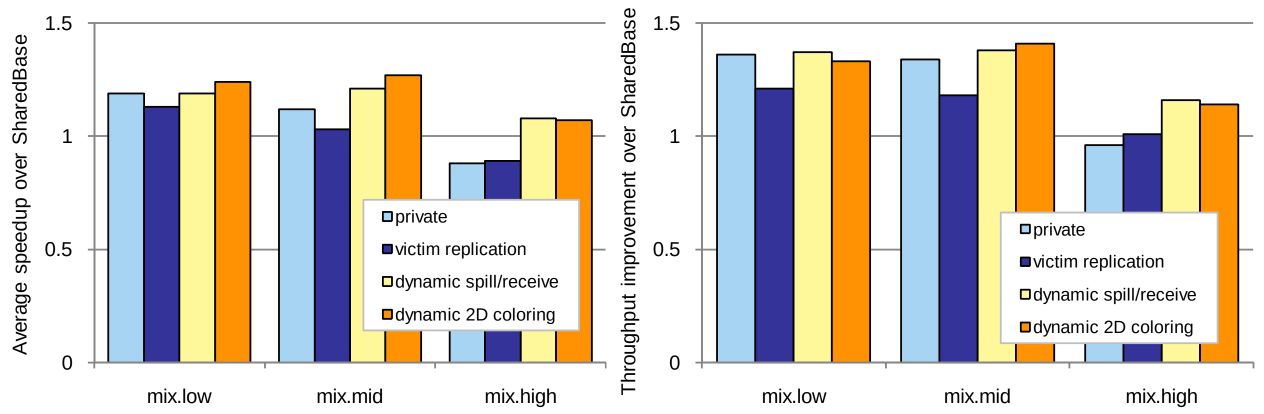


Figure 14: Performance of multiprogrammed workloads under the private cache scheme, victim replication scheme, dynamic spill and receive scheme , and dynamic 2D page coloring scheme normalized to the baseline shared cache scheme.

scheme. In comparison with the private cache scheme, the dynamic 2D scheme achieves an improvement of up to 14.3% in average speedup and 18.8% in throughput in the best case. For "mix.low" and "mix.mid" workloads, **Dyn2D** outperforms **DSR** since it gains benefit from balancing cache contention. For "mix.high", the advantage of cache conflict reduction diminishes as cache contention becomes very high. As a result, these two schemes have very close performance.

The dynamic 2D page coloring scheme has a very slight throughput degradation than the private cache scheme for the "mix.low" workload. This is because the programs in this workload have low cache space requirement each and can run efficiently on a private cache. On the other hand, the dynamic 2D page coloring scheme introduces some interferences in allocating pages and accesses. An interesting observation is that the dynamic 2D page coloring scheme performs best for the "mix.mid" workload in both metrics. This is because this workload provides the largest room for trade-off between miss rate and access latency, and the dynamic 2D page coloring scheme is able to hit the right point in the trade-off span.

4.4 SUMMARY

In this chapter, the problem of managing distributed L2 caches on a large-scale chip multiprocessor to achieve a high single-threaded program performance is studied. First, a static 2D page coloring scheme is studied, which utilizes static trace information to provide a near-optimal upper bound. Then a dynamic 2D page coloring scheme is proposed as a practical solution. The dynamic page coloring scheme utilizes a heuristic strategy to balance both cache miss rate and cache access latency based on the current cache usage information. Due to the lack of comprehensive information, the initial data placement can lead to hot-spots later on. For this reason, the page migration technique is proposed to mitigate the impact of hot-spots. The evaluation shows improved performance.

The outline and achieved contributions of this study are summarized as follows:

- A static off-line 2D page coloring algorithm is proposed and studied. It assigns a memory page to a cache bin based on detailed page conflict and access frequency information. The automated process tunes key algorithm parameters to trade cache access latency over cache miss rate and vice verse. As a result, the near-optimal performance of the static data mapping strategy is obtained. It presents a relative tight bound on the performance of the shared cache structure when the cache hit latency and the on-chip cache miss rate are optimized together via a flexible data mapping scheme.
- A dynamic online algorithm is proposed and studied. It maps pages to L2 cache bins and migrates previously mapped pages. The proposed algorithm uses only run-time information about cache bin hotness and page usage when selecting a target cache bin for a new page. The design and implementation issues are discussed in detail.
- The proposed schemes are evaluated and compared with the existing shared and private cache schemes. The quantitative study shows that the proposed schemes achieve higher performance because they balance cache miss rate and cache access latency effectively. The proposed schemes are shown to be relatively insensitive to the scale of the CMP and the size of the cache slice. The proposed dynamic 2D coloring scheme achieves much of the performance potential identified through the limit study using the off-line algorithm.

5.0 DATA AFFINITY ANALYSIS FOR MULTITHREADED PROGRAMS

Improving data affinity for multithreaded programs is not as straightforward as single-threaded programs. The shared data are accessed by multiple threads simultaneously. Thus it is difficult to place the data in a single location while achieving lowest access latency for all sharers. As a result, optimization techniques such as data replication seem necessary. On the other hand, private data are used mostly by a single thread. It is desired to place a block of private data into the L2 cache slice of its owner. Data replication and migration techniques can remedy the performance loss caused by blind data distribution. However, data replication and migration incur extra data placement and movement in the L2 cache. These activities lead to increased cache pressure and can have negative impact if their engagement is not carefully controlled. To avoid these issues, it is important to make informed decisions when data are placed initially and moved later on.

To improve the data affinity of latency-oriented multithreaded programs, the approach proposed and studied in this chapter uses affinity hints generated from off-line analysis to guide data distribution at run time. The hints should be sufficiently general so that they can be used by multiple runs with different input sets and micro-architecture configurations. To achieve this, a fundamental understanding of the general behavior of latency-oriented multithreaded programs is necessary.

In this chapter, we first analyze the general cache access behavior of a set of latency-oriented multithreaded programs. Then the most commonly seen memory access patterns are summarized. The programs are drawn from the widely used SPLASH-2 benchmark suite [86] and the PARSEC benchmark suite [76] and are listed in Table 2 of Chapter 3. Based on the qualitative study, the proposed memory access pattern recognition algorithm is introduced, followed by the evaluation results.

5.1 PROPERTIES OF MULTITHREADED PROGRAMS

5.1.1 Parallel Programming Models

Most multithreaded programs can be categorized into two types in terms of the programming model they use: the *throughput-oriented model* and the *latency-oriented model.* The throughput-oriented model is commonly found in server programs. In these programs, a master thread is responsible for picking up tasks from a task queue and distributing them to slave threads for parallel execution. Tasks share similar characteristics. But the jobs they process are mostly independent on each other. There is little communication and synchronization among slave threads. In this type of program, performance is measured by the number of transactions or tasks completed in a given unit of time. The throughput of the whole server program is considered more important than the execution time of individual tasks.

The latency-oriented model encompasses a broad range of multithreaded programs, especially scientific workloads and kernel-based programs [7]. In these programs, a large and complex problem is partitioned into subproblems to solve them in parallel. In contrast to throughput-oriented multithreaded programs, latency-oriented programs are more concerned about the wall-to-wall execution time. During a program's execution, there may be a lot of data exchange and synchronization from time to time, depending on its algorithm design. Latency-oriented multithreaded programs are the main focus of study in this chapter.

5.1.2 Data Structure Types

A program's data access patterns are highly dependent on the associated data structures. Important data structures are described here before introducing the access patterns. Through the examination of programs from the SPLASH-2 benchmark suite, it is easy to reach a conclusion that the commonly used data structures are array, linked list, tree, graph and their compositions. These data structures are indeed the most important ones you can find in almost any programs and programming language textbooks.

Among these data structure types, the array plays an especially important role when designing a program because it is easy to use and maintain. An array can be assigned statically at compile time or allocated dynamically at run time when the size is known. An array element is accessed directly with its index. Since an index is often calculated explicitly from dependent variables, the array's access pattern is often predictable given the index calculation formula. In contrast, locating a node in a linked list usually requires a traversal of the list starting from the head. So the nodes near the list head tend to be accessed more frequently than those at tail. The dynamic expansion and shrinking of the linked list can also lead to a non-contiguous memory layout, posing a challenge for off-line analysis. Tree and graph are more advanced data structures. Since a node can spawn multiple children in a tree or a graph, the search path becomes less predictable than a linked list. The irregularity makes it impossible to recognize their access patterns. However, the rule of thumb is that the nodes close to the root of the structure are accessed more often than others. Fortunately, the tree and the graph are used less frequently than the array in the latency-oriented multithreaded programs we examined.

5.1.3 Memory Allocation Types

A memory area touched by a program can be categorized as either a *static data region* or a *dynamic data region*. Global variables and data structures are those used to track program-wide information, to synchronize and to exchange shared data among threads. They are often assigned statically. The location and size of the static data are known prior to a program's execution. The determinism makes off-line affinity analysis for static data relatively easy. The size of static data usually does not scale when the input set changes. Thus the ratio of static data access to all data access can decrease as the input set becomes larger.

The dynamic data plays an important role in large-scale parallel programs. From time to time, it may require to allocate memory areas dynamically at run time, since the variations of a program's input can prohibit it from claiming a memory area statically without knowing the data size. *malloc* is a typical library function to provide dynamic allocation of memory region in the C language. In this study, *malloc* is used as an indication of a dynamic allocation in general. Dynamic data are usually the target of computation, thus they tend

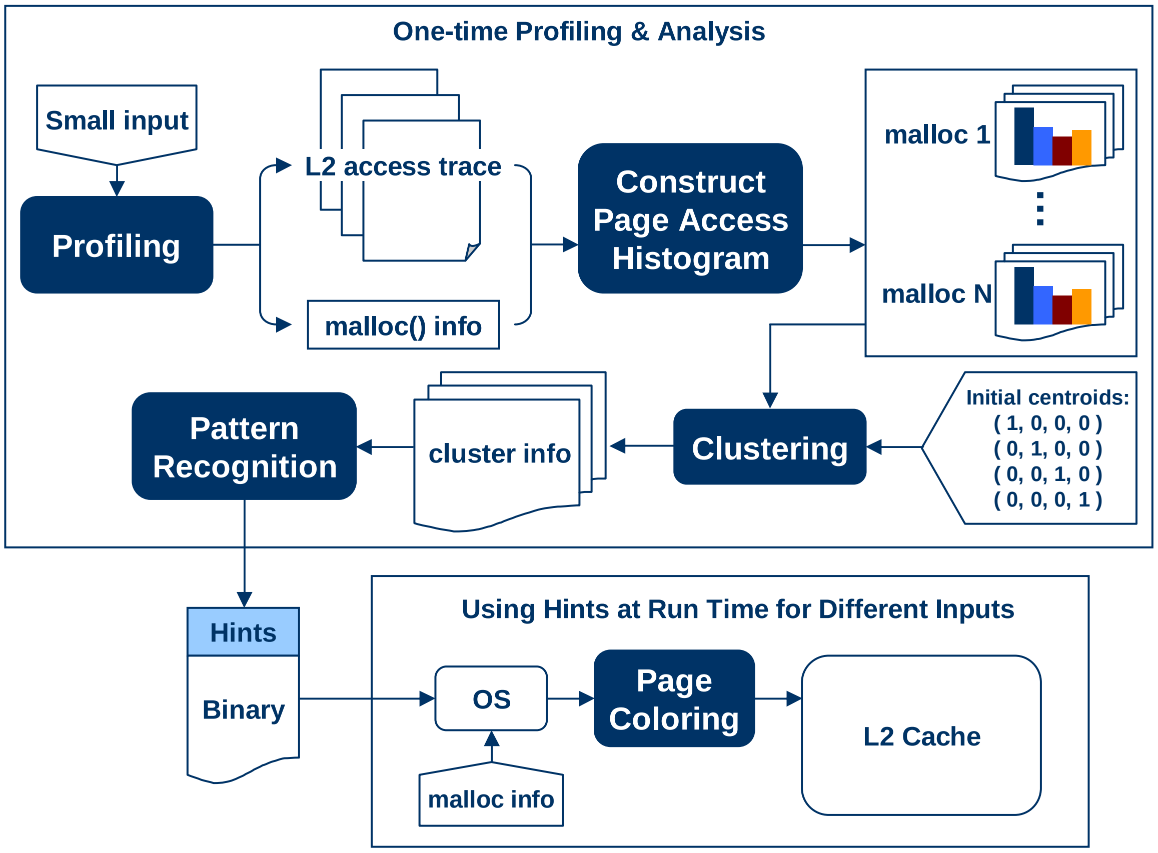


Figure 15: The one-time profiling and data access pattern analysis flow (upper box) and the hint exploitation flow (lower box) of the proposed hint-guided data placement scheme.

to be accessed more frequently than static data. Careful distribution of dynamic data can have a positive impact on the performance of a latency-oriented multithreaded program.

5.2 BASIC APPROACH

Figure 15 depicts the two major stages of the proposed hint-guided data placement scheme: the one-time profiling and data analysis phase and the hint exploitation phase. In the first phase, the proposed scheme profiles a given program's L2 cache accesses with a test input. Based on the collected traces, access histograms are constructed for each page (buckets in each histogram count accesses from different processors). Then K-means clustering algorithm is applied on those per-page access histograms to derive page clusters. Given the page cluster information, our scheme finally determines the patterns for dynamic and static data areas and attach those hints to the binary. At the run-time stage, the OS peels off the hints from the binary and uses the information whenever a new memory page mapping event occurs. In this section, we first examine the common data access patterns found in multithreaded applications. We will then discuss in detail the data access pattern recognition algorithm and how the resultant data affinity hints are exploited to guide the OS cache management decisions.

5.2.1 Access Pattern Classification

In this section, some of the observed memory allocation and access patterns are summarized. Access patterns for static data area and dynamic data area are introduced separately.

A program's static data access patterns can be broadly characterized as three types:

- *Shared*: Data structures of this type are accessed by all threads. They can be evenly shared by all threads all the time. Or they can be accessed by only one or more threads at a time. But as the program phase changes, they become dominantly accessed by other threads. Over the period of the program's whole execution, it is difficult to identify patterns. Global data structures usually fall into this category.
- *Local*: Data structures of this type are only accessed by one thread. For instance, a block of data allocated within a program's parallel section usually falls into this category. Its existence is only aware by the thread which allocates it. So it can only accessed by that thread.
- *Private*: Data structures of this type are shared by all threads, but they are mostly accessed by a single thread. For instance, a global data structure is mostly used by the main thread even though all threads have periodic accesses to it.

Benchmarks like *radiosity*, *fmm*, *raytrace*, *water-nsquared*, and *water-spatial* have a significant portion of accesses to static data structures, especially when the input set is relatively small. Most programs introduce many more accesses to dynamic data structures than to static data structures when the program input set grows. In addition, the access patterns of static data are relatively easy to identify. So more attention will be paid to the access patterns of dynamic data from now on.

Access patterns for dynamic data are more complex. They are categorized into types of *even partition*, *scattered*, *private*, *small-entity*, and *shared*. They are examined type by type in the following paragraphs.

Even: In many scientific computation programs, a large one-dimensional data array is allocated at the beginning of the execution and initialized with input data. A data array can be easily partitioned among threads due to its regularity. The array index is commonly a function of the thread ID and some loop indices. As a result, a data array can be unambiguously partitioned in terms of the thread ID. For example, the following pseudo-code illustrates how an array is allocated by the main thread and then how each thread accesses its own partition by using its thread ID (*ProcNo*).

```
Main Thread:
/* array allocation */
Array = malloc(sizeof(int) * NumProc * N);

Thread [ProcNo]:
/* a partition of the array is accessed
   by the corresponding thread */
for(i = 0; i < N; i++)
    Array[ProcNo * N + i] = i;
```

This access pattern leads to even partitioning of the whole data array among threads. It presents a good optimization opportunity to distribute data in the L2 cache according to the data partition. As shown in Figure 16(a), the access pattern of the example pseudo-code can evenly split an array into four partitions. Each partition is exclusively accessed by one thread in order. Hence, to achieve ideal data affinity, partition 0 can be placed in the local cache slice of thread 0, partition 1 of the same array can be placed in the local cache slice of thread 1, and so on. Sometimes, accesses to a data array can be interleaved in a finer

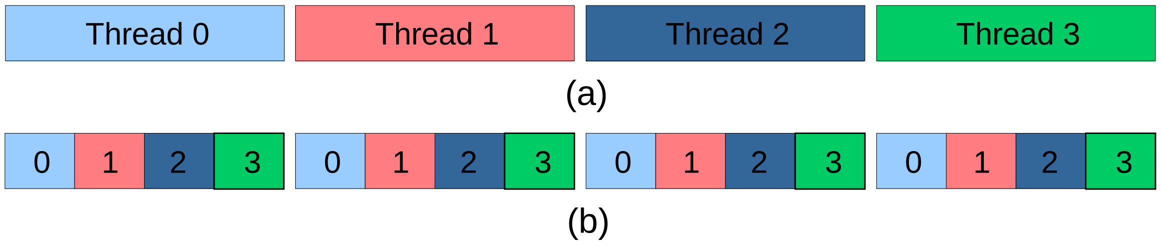


Figure 16: An example of a dynamically allocated memory area shared by four threads evenly.

granularity depending on the style of the loop iteration. The following code illustrates such case:

```
Thread [ProcNo]:
for(i = 0; i < N; i++)
    Array[i * N + ProcNo] = i;
```

The corresponding data access pattern is shown in Figure 16(b). As you may note, the new pattern is essentially the same as the previous one, except the data layout is interleaved. The compiler can always reschedule the data layout and transform it back to the pattern in Figure 16(a). Among the benchmarks examined, *fft*, *lu* and *blackscholes* have this kind of data access pattern.

Scattered: At times, the workload assigned to each thread is not balanced. So a thread wants to manage the data storage allocation by its own. That is, the data allocation is done within each thread instead of in the main thread. The programmer can also choose a separate memory area to allocate for each thread. Hence, unlike the "even" pattern, the whole data set is scattered into multiple memory regions. These memory areas are not necessary continuously allocated. The following pseudo-code demonstrates a typical situation where the scattered pattern emerges.

```
Thread [ProcNo]:
Array = malloc(sizeof(int) * N);
```

```
for(i = 0; i < N; i++)
    Array[i] = i;
```

Since the allocated memory area is exclusively accessed by its owner, the optimal data distribution is straightforward. The allocation can also be done in the main thread, especially when a two-dimensional array is used. The following pseudo-code demonstrates the described scenario:

```
Main Thread:
ArrayPtr = malloc(sizeof(int) * NumProc);
for(i = 0; i < NumProc; i++)
    ArrayPtr[i] = malloc(sizeof(int) * PartitionSize[i]);

Thread [ProcNo]:
for(i = 0; i < PartitionSize[i]; i++)
    ArrayPtr[ProcNo][i] = i;
```

Note how *ProcNo* plays a role in addressing array elements. Data placement of this pattern can use *ProcNo* as a hint since those in-order allocated areas are exclusively accessed by each thread using *ProcNo*. Because two-dimensional arrays are extensively used, this access pattern is very common in the multithreaded programs examined in this study. The representative programs that more or less exhibit this pattern are *barnes*, *cholesky*, *fmm*, *ocean*, and *radix*.

Private: There are occasions when a shared data area is mostly accessed by only one thread. These memory areas are commonly allocated for auxiliary structures in the main thread. They help record temporary information while initialization progresses. These areas are regarded as private because it is logical to place these data in cache slices, where they are accessed the most. An example is the program *radiosity*, which has a global data structure accessed a lot by the main thread.

Small-Entity: When a program's data are organized using a linked list, a tree, or a graph, dynamic memory allocation is often required to save storage space. It involves intermittent allocation and freeing of nodes. A small trunk of memory area can be repeatedly allocated and reclaimed by multiple *malloc* and *free* instances. This behavior poses extreme difficulty for tracking memory usage and managing data at coarse granularity. Since the cache management scheme studied in this thesis relies on the page mapping mechanism to control data placement, it is impossible to differentiate two nodes co-located within the same page. *cholesky*, *raytrace* and *swaption* are typical programs with this small-entity pattern. Another representative case involves the small-entity pattern is the usage of data stack. On function calls and returns, the data stack expands and shrinks accordingly, creating complex usage patterns. However, unlike the previous case, the ownership of the stack data is explicit. This is because data items in the stack are used as function parameters and local variables, which are certainly private to the accessing thread.

Shared: The last category contains all data areas that could not be classified to any type mentioned previously. These areas are highly shared by multiple threads. No particular affinity pattern can be found in these areas. The pattern can also change under different situations. However, data areas can be loosely separated into read-only sharing and read-write sharing. Because the input variation seldom affects the read/write behavior of the data, it is safe to mark data areas with read-only sharing property as replication candidates.

Other more complex memory access patterns are revealed in the studied programs. But this study does not attempt to recognize all possible memory access patterns that may prove useful. By focusing on the most frequently observed access patterns presented in this section, it tries to motivate the proposed software-oriented cache management approach.

5.2.2 Access Pattern Recognition Algorithm

Efficient recognition of a program's memory access patterns is not a trivial task. By "pattern", it means an abstraction that can effectively capture a program's cache access behavior under different conditions. The goal is to derive data allocation hints that can be used across different program parameters and microarchitecture configurations, such as different cache sizes. In addition, choosing a proper method to represent these hints is critical in this scheme. For the static data whose layout is determined at compile time, a hint can be given as a plain virtual page number to cache slice mapping. However, the location and size of a dynamic data area are unknown until the corresponding *malloc* returns at run time. There is no way to directly command a page's placement solely based on off-line information.

Our strategy is to associate one dynamic hint with each *malloc* instance, which can be identified uniquely by the file name, the line number in source code, and the number of times it has been called. A dynamic hint only expresses which pattern the memory area allocated by the corresponding *malloc* instance would exhibit, instead of giving specific placement information. Only at run time, when the address and size of a dynamic area are determined, the actual page to cache slice mapping is generated. The hints can be embedded in a program's binary and loaded into the OS whenever it is necessary. When a page fault occurs, hints are consulted to select a page location among all L2 cache slices. In what follows, pattern recognition algorithms for both dynamic data and static data are introduced.

5.2.2.1 Pattern Recognition for Dynamic Data To identify the access pattern for a dynamically allocated memory area through L2 cache access trace analysis, it is required to profile the program with a small, reasonably representative input set once. During the profile execution, an L2 cache read trace is collected from each tile. In addition, the starting address and the range of every *malloc* system call are recorded. Then each access trace item from the trace is processed by checking it against all *malloc* ranges. An L2 cache access is a dynamic access if it falls into one of the *malloc* ranges. In order to count the access frequency, each page within the *malloc* range is associated with a counter vector. When a page receives an access from a tile, the counter corresponding to that tile is incremented. After all traces have been processed, the counter vector of a dynamic page represents the histogram of accesses from all tiles. The counter vector is normalized to the maximum counter value within it. Then *K-means clustering* method is applied on all counter vectors associated with the corresponding pages of a given *malloc* range. The initial centroids for this clustering method play an important role in determining what the final clusters look like. In this study, the initial centroids are carefully designed so that pages accessed exclusively by different threads are grouped into different clusters. An example of the initial centroids used to cluster traces from a 4-tile CMP is:

```
C0 (1, 0, 0, 0)
C1 (0, 1, 0, 0)
C2 (0, 0, 1, 0)
C3 (0, 0, 0, 1)
C4 (1, 1, 1, 1)
```

The *K-means clustering* algorithm is sketched as follows:

```
do {
    1. Assign all vectors to their nearest
       cluster centroids based on Euclidean distance.
    2. Determine the error as the distance between the
       vector and its nearest centroid as the error.
    3. Accumulate the total error for all vectors
    4. Update the new cluster centroids by averaging
       vectors within each cluster.
    5. Calculate the error difference between
       the current iteration and the last iteration.
} while(error difference > threshold)
```

After the clustering procedure finishes, each cluster contains many vectors corresponding to pages from the same dynamic memory area. All pages from cluster 0 (*C0*) are accessed mostly by tile 0. All Pages from cluster 1 (*C1*) are accessed mostly by tile 1. This applies to all clusters, except the last one (*C4*), where pages are accessed almost equally by all tiles.

Next, patterns discussed in Section 5.2 are identified based on the derived cluster information. The even partition pattern is checked by counting the number of "fitting pages" in each cluster. For instance, suppose a dynamic memory area has 16 pages and traces are profiled on a 4-tile CMP. Figure 17(a) shows a typical even partition pattern, where each partition has 4 pages. An example of the clustering result is given in Figure 17(b). As illustrated, the partitions may not be perfectly even in practice. Some clusters can have more pages while others have less pages than the average. To examine if a memory area has even partition pattern, its clustering results (*i.e.*, partitions) are compared with the even

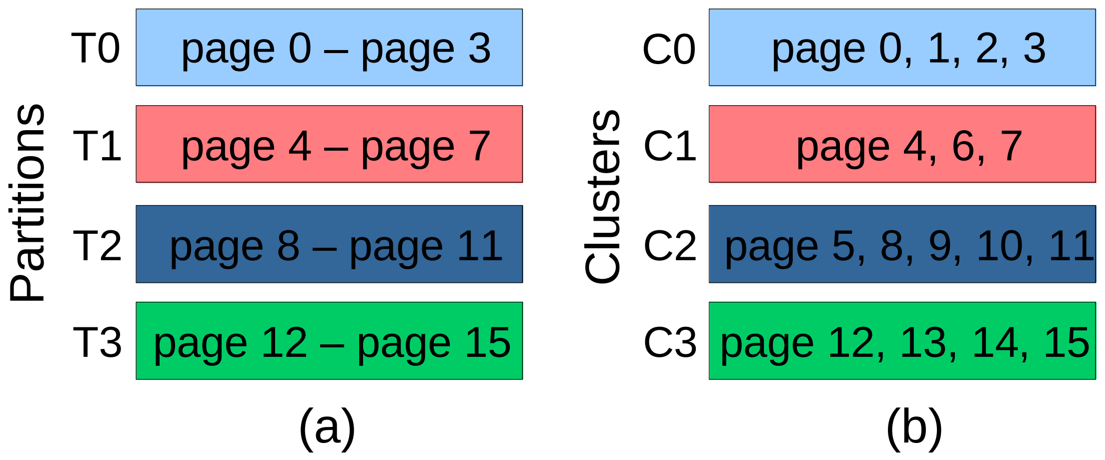


Figure 17: (a) An example of the even partition pattern. (b) Page clusters after K-means clustering of the access histograms.

partition pattern for this memory area in an ideal situation. In this example, pages in *C0* are compared to pages in *T0*, pages in *C1* are compared to pages in *T1*, and so on. A cluster is defined as a *fitting cluster* if half of the pages from the ideal partition range reside in the cluster. A *malloc* is regarded as having an **Even Partition Pattern** if more than 75% of all clusters are fitting clusters.

Recognition of the scattered pattern follows a similar procedure, except that the clustering method has to be performed for all instances of the same *malloc*. Figure 18 shows an example of a scattered pattern where the *malloc* is called 4 times within a *for* loop. The aggregated space of these 4 *malloc* instances occupies 16 pages. The area allocated by the first *malloc* instance is mostly accessed by tile 0. The area allocated by the second *malloc* instance is used mainly by tile 1 and so on. In order to examine if a *malloc* exhibits scattered pattern, all instances of the same *malloc* are checked along with the information about the sequence they are called. If an instance is the **N**th call of the same *malloc*, the sequence number of this *malloc* instance is **N - 1**. If half of the pages allocated by a *malloc* instance are assigned to one cluster, the cluster number is checked against the sequence number of the *malloc* instance. If more than half of the instances of the same *malloc* have cluster number and sequence number matched, this *malloc* is regarded as having an **Ordered Scattered Pattern**.

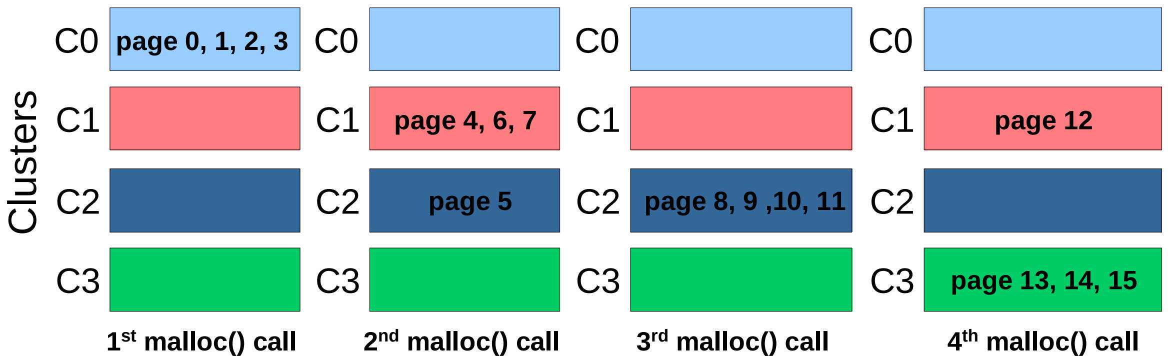


Figure 18: An example of the clustering results of four instances of the same *malloc*.

If it does not have **Ordered Scattered Pattern**, the *malloc* instances might be called within the parallel threads. In such a case, the *malloc* is defined as **Private Scattered Pattern**. If majority of the pages allocated by a *malloc* are clustered into one cluster, the *malloc* is defined as having **Private Owner Pattern**. Lastly, when pages are grouped into the last cluster (*C4*), they are shared by all threads. It cannot differentiate which thread accesses the pages more than the others. The corresponding *malloc* does not associate with any recognizable pattern. It is marked as a **Shared Pattern**.

5.2.2.2 Affinity Hint for Static Data

Compared to the pattern recognition method for dynamic data, analysis for static data is relatively easy, since it is assumed that the access pattern for a static page is deterministic. A simple page number to cache slice mapping can be used as a hint. The hint generation is straightforward. If the number of accesses from a particular tile counts more than 50% of total accesses for a page, the page is assigned to that tile. Otherwise the page is marked as **Shared**. The threshold value 50% essentially controls if a page should be treated as a shared page. A higher threshold value would result in a more conservative page placement decision. There is a lower bound for the threshold in terms of the number of tiles in a CMP. For a 16-tile CMP, the threshold value should be greater than 6.25% since that is the average share each tile should receive. In this study, the evaluation results show that 50% works well.

5.2.3 Hint-Guided Data Placement

After various memory access patterns are recognized for static and dynamic memory areas, an efficient hint representation method becomes necessary so that these patterns can be utilized by later executions. Since hints for dynamically allocated data and static data are different, they are presented separately. A static hint gives the target cache slice for a static page explicitly, thus can be used directly at run time. A dynamic hint only tells the type of a *malloc* instance in the source code. It needs to be translated into actual mappings when the starting address and the size of a memory area are determined by the corresponding *malloc*. This section discusses how to utilize these dynamic pattern hints at run time to guide data placement.

The OS reads carried hints when loading a program for execution. Each hint is represented by the OS internally using a data structure such as following:

```
struct DynamicHint {
    Integer MallocID
    Integer HintType
    Integer Counter
    Integer Target
}
```

When *malloc* is called, the corresponding **MallocID** (*e.g., a combination of the containing file name and the line number*) is searched against the hint list. If there is a match, a new pattern descriptor is created. The following pseudo-code describes how a dynamic hint is translated into a pattern descriptor at the time of *malloc*:

```
struct PatternDescriptor {
    Integer PatternType
    Addr_t Start
    Addr_t End
    Integer Target
    Pointer Next
}
```

```
MallocHandler()
{
    normal malloc operation
    ......
    ID = current malloc identifier
    Start = start address of the allocated memory
    End = end address of new allocated memory
    ThreadID = the calling thread ID

    for each hint item h in hint list
        if ID equals h.MallocID
            p = new PatternDescriptor structure
            p.PatternType = h.HintType
            p.Start = Start
            p.End = End

            if p.PatternType is OrderScattered
                p.Target = h.Counter
                h.Counter++
            if p.PatternType is PrivateScattered
                p.Target = ThreadID
            if p.PatternType is PrivateOwner
                p.Target = h.Target

            add p into the pattern list
            exit the loop

    return start address
}
```

The following text describes how our scheme generates target tile ID (*i.e., where the page is placed in the cache*) from pattern descriptors at run time.

Even Partition Pattern: In the above **MallocHandler()**, only the whole memory range is generated for the even partition pattern. The actual data placement decision is deferred until a page fault is triggered. The whole memory range is artificially partitioned into 16 equal sub-ranges. Then the triggering virtual address is checked to find out which range it falls in. The corresponding sub-range index is the target tile ID. The following pseudo-code illustrates this process:

```
PageFaultHandler() {
    VA = virtual address that triggers the page fault

    /* the default value -1 indicates a shared pattern */
    TargetTID = -1
    for each pattern p in the pattern list
        if VA falls in the range [p.Start, p.End]
            if p.PatternType is EvenPartition
                PartitionSize = (p.End - p.Start) / NCore
                TargetTID = (VA - p.Start) / PartitionSize
    save TargetTID into the corresponding page table entry
}
```

Ordered Scattered Pattern: As described in **MallocHandler()**, the target tile ID is determined during the process of translation from the hint to its pattern descriptor. A counter with an initial value of 0 is associated with the hint structure. Each *malloc* call uses the counter value as the target tile ID and increments the counter. For example, if the counter value in the hint structure for a *malloc* is 2, then the memory area returned by the next call to this *malloc* is assigned to the cache slice in tile 2. The counter value is then increased to 3. Given the pattern descriptor, the following pseudo-code illustrates how tile ID is determined:

```
PageFaultHandler() {
        VA = virtual address that triggers the page fault

        /* the default value -1 indicates a shared pattern */
        TargetTID = -1
        for each pattern p in the pattern list
            if VA falls in the range [p.Start, p.End]
                TargetTID = p.Target
        save TargetTID into the corresponding page table entry
    }
```

Private Scattered Pattern: If a hint indicates that a *malloc* has this pattern, the memory area returned by this *malloc* is placed directly to the tile, who calls this *malloc* instance. The implementation of **MallocHandler()** is the same as **Ordered Scattered Pattern**.

Private Owner Pattern: For this pattern, a target tile ID comes with the pattern type in the hint explicitly. The allocated dynamic area is simply placed in the specified target cache slice. The implementation of **MallocHandler()** is the same as **Ordered Scattered Pattern**.

Shared Pattern: As shown in **MallocHandler()**, when there is no match in the pattern list, -1 is assigned to the target tile ID to indicate a shared pattern for a page. By default, no effort is made to optimize for a dynamic data area of this pattern. Since the data in a page are shared by all threads, placing this page in any single cache slice would cause uneven access latency among all sharers. It can also create a hot-spot in the hosting cache slice. As a result, it seems reasonable to distribute the data in this page at cache block granularity instead of page size to balance L2 cache accesses among all cache slices. Moreover, it is beneficial to combine other hardware-based techniques to improve access latency for highly shared data since shared memory accesses account for a significant portion of all L2 cache accesses. To capture temporal data affinity for shared data, one can benefit from the victim replication scheme [111]. As the private data are correctly placed, the amount of data replication and the number of resulting conflict misses can be reduced significantly. Other hardware techniques such as *multicast* can also be employed. The motivation here is that highly shared data are likely possessed by multiple tiles simultaneously in their L1 caches. If the load request misses in the local L1 cache, it is very likely to find the data in one of its neighbors' L1 caches. Based on this observation, a multicast scheme can work as follows: A read miss leads to an access to the home directory to find out where to fetch the data. On its way to the remote directory, L1 caches of all visited tiles are checked for the data. If a copy of the data is found, it is replied immediately to the requester, which could resume execution earlier. A similar multicast scheme was recently proposed in [19], albeit in a different context (private L2 cache).

5.3 EVALUATION RESULTS

Five cache schemes are evaluated and compared in order to demonstrate the benefit of the proposed data placement scheme. These five of them are: the shared cache scheme (**L2S**), the private cache scheme (**L2P**), the victim replication scheme (**L2VR**), the hint-guided data placement scheme (**L2H**), the hint-guided data placement and data replication scheme **L2HR**. For **L2HR**, the page placement in the L2 cache for private data is guided by **L2H** as usual, but data replication is used to save remote access latency for shared data blocks. The data replication is also guided by the derived hints.

5.3.1 Hint Accuracy

Figure 19 shows the L2 cache access distribution according to the type of access pattern identified by the proposed pattern recognition algorithm. In case of *cholesky*, the algorithm recognizes around 15% of total accesses belonging to *private scatter* pattern. Nearly 80% of total accesses come from shared data areas. These numbers are consistent with the curves in Figure 1 shown in the introduction section. Other benchmarks such as *raytrace*, *volrend*, *water-nsquared* and *water-spatial* also have very large number of L2 accesses for shared data. This is aligned to the data structures used by these programs—they naturally have a lot of data sharing. In contrast, *fft*, *lu*, *ocean*, *radix* and *swaptions* exhibit abundant private

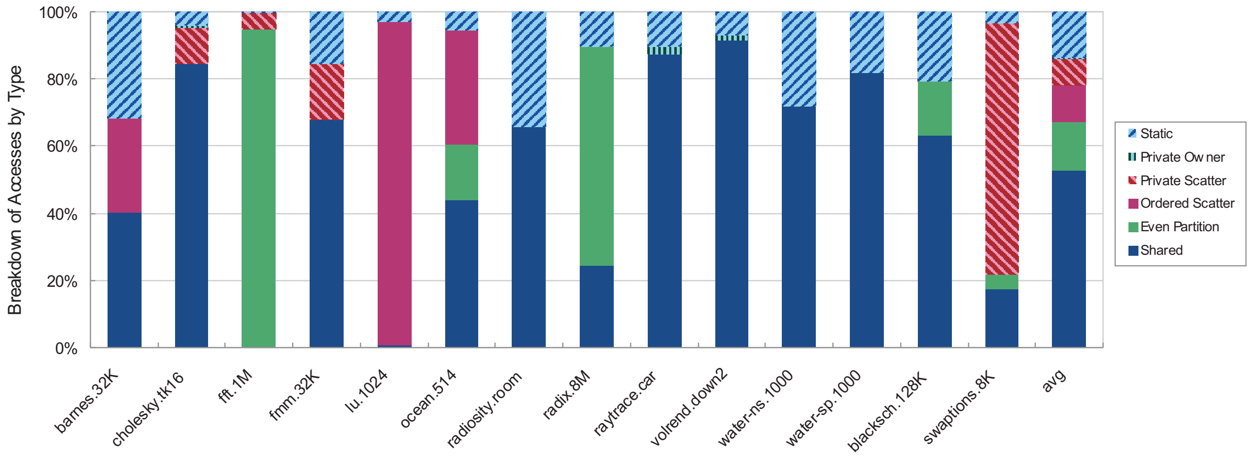


Figure 19: Breakdown of L2 cache accesses based on the classified pattern types (median input set).

data accesses which are captured by the identified patterns. Overall, the recognized patterns capture more than 50% of total L2 accesses.

Figure 19 shows the algorithm is effective in recognizing different types of access patterns. It is interesting to see the accuracy of these identified access patterns. To do so, the access histogram of all pages and their locations suggested by the hints during the simulation are collected. The data affinity hint is considered "accurate" and a page is considered "accurately placed" if the local tile of a page predicted by the hint accesses it the most. The *hint accuracy* is defined as the ratio of the number of accurately placed pages to the total number of placed pages. Note that the shared pages are not considered in this calculation. The accuracy metric measures how good the algorithm is at identifying and representing the access patterns. Another metric defined is called *coverage*, which is the ratio of the number of accesses to the "accurately placed" pages to the total number of L2 cache accesses. This metric includes the impact of the shared pages. It indicates how effectively the affinity hints can cover all L2 cache accesses. Table 3 shows the results with the small and median input sets. The derived hints achieve a high accuracy of over 80% in most cases. The exceptions

Program	*Small*		*Median*		*Large*	
	Accuracy	*Coverage*	*Accuracy*	*Coverage*	*Accuracy*	*Coverage*
barnes	82.1%	47.8%	84.6%	43.3%	83.1%	45.6%
cholesky	82.9%	7.3%	85.9%	9.0%	84.6%	9.2%
fft	96.1%	53.7%	99.0%	69.4%	98.0%	65.7%
fmm	88.2%	28.1%	90.2%	28.5%	90.1%	29.2%
lu	96.7%	77.1%	98.3%	87.4%	98.4%	87.0%
ocean	99.0%	48.9%	98.7%	52.5%	98.3%	50.1%
radiosity	97.7%	26.8%	96.6%	33.5%	94.9%	35.7%
radix	90.4%	69.0%	66.1%	54.3%	63.7%	57.8%
raytrace	68.4%	7.9%	31.7%	3.9%	34.5%	6.1%
volrend	80.4%	9.7%	79.6%	8.0%	80.1%	8.9%
water-ns	45.0%	25.2%	45.0%	25.7%	45.1%	26.0%
water-sp	67.2%	16.6%	67.2%	17.2%	68.5%	18.5%
blackscholes	83.7%	34.2%	60.6%	29.8%	65.3%	34.1%
swaption	60.8%	44.8%	61.7%	47.3%	60.8%	46.4%

Table 3: Pattern recognition accuracy and coverage.

are *raytrace*, *water-ns*, and *water-sp*. These programs use complex data structures such as trees and 3D matrices during computation. Data structures are updated constantly, leading to many *malloc*s and *free*s, which poses a challenge for the off-line pattern analysis scheme.

The coverage varies from one program to another. Some programs such as *lu*, *ocean*, and *radix* have good data affinity and well recognized data partitions, thus achieving a high coverage ratio. Others such as *cholesky*, *raytrace*, and *volrend* have low coverage. This means the cache access patterns of major data regions are not recognized since they are widely shared by threads. Interestingly, the accuracy and coverage measure of small input set and those of median input set are very close. This shows that the provided hints are stable across different input sets. In some cases, the accuracy and coverage become even higher for the median input set even though the hints are derived from the small input set. This can happen because the target data area becomes larger with the median input set, capturing relatively more accesses. The results for the large input set are very close to these of the median input set.

5.3.2 Performance Improvement

Let us turn the attention to the program performance and behavior on the studied machine architecture. Figure 20 shows the normalized execution times of the five cache management schemes for the small, median and large input sizes as given in Table 2. Execution time is normalized to **L2S**, the baseline design. Since the hints are derived from small input, Figure 20(a) provides a measure of how the hints perform under ideal situation. Comparing the results in Figure 20(a), (b) and (c), it is obvious that the relative performance of these schemes change little with different input sizes. One major reason is that many of these examined benchmarks are well optimized and have relatively small working set sizes that do not scale with the benchmark input. In other words, scaling up the input size does no cause the situation, where the working set cannot fit in the cache slice. This explains why **L2P** performs better than **L2S** most of the time. The results also demonstrate that the off-line pattern recognition algorithm performs robustly for different input sizes as **L2H** consistently provides around 10% performance improvement over **L2S**. This shows that the proposed algorithm captures existing program access patterns effectively. The patterns are stable and help place memory pages correctly even with changed input. In the following discussions, results with the median input set are used unless otherwise noted.

It is worth looking at Figure 20(b) in more detail. First, **L2P** performs considerably better than **L2S** due to the small program working set size. The exceptions are *cholesky*, *ocean*, *raytrace* and *volrend*, where data sharing is relatively high. For the same reason, **L2VR** is also very effective, achieving around 11% execution time improvement over **L2S** and similar to **L2P** on average. **L2VR** brings the L2 cache access latency close to **L2P** through replication. **L2H** approaches the performance of **L2P** and **L2VR** very well, achieving 10%

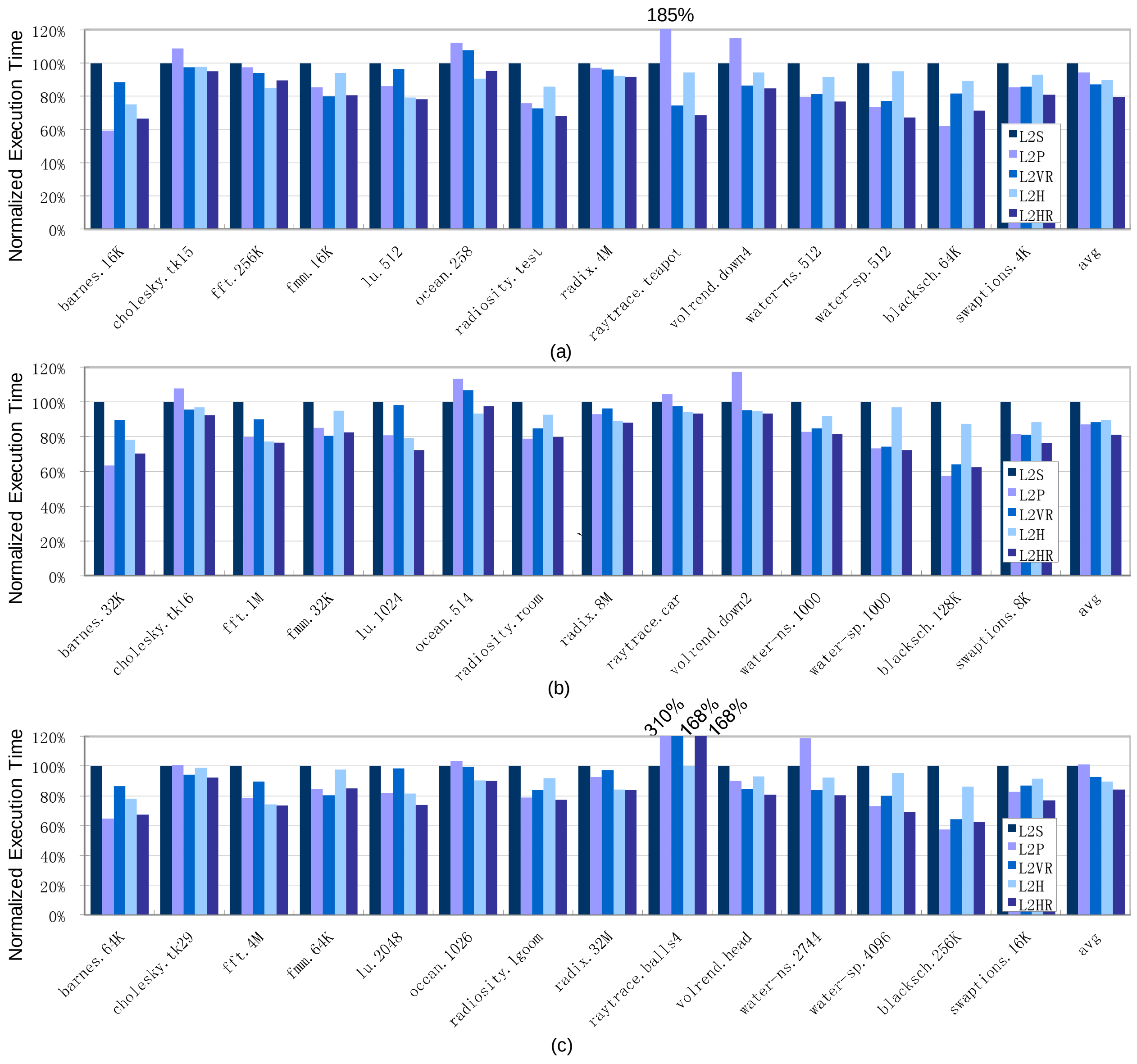


Figure 20: Execution time of benchmarks with (a) small, (b) median, and (c) large input set, normalized to the execution time of the shared cache scheme (**L2S**).

execution time improvement over **L2S**. The improvement of **L2H** comes from the optimized data affinity for those data used mostly by one thread. No effort is made by **L2H** to tackle highly shared data. When hints are used to direct data replication, **L2HR** improves

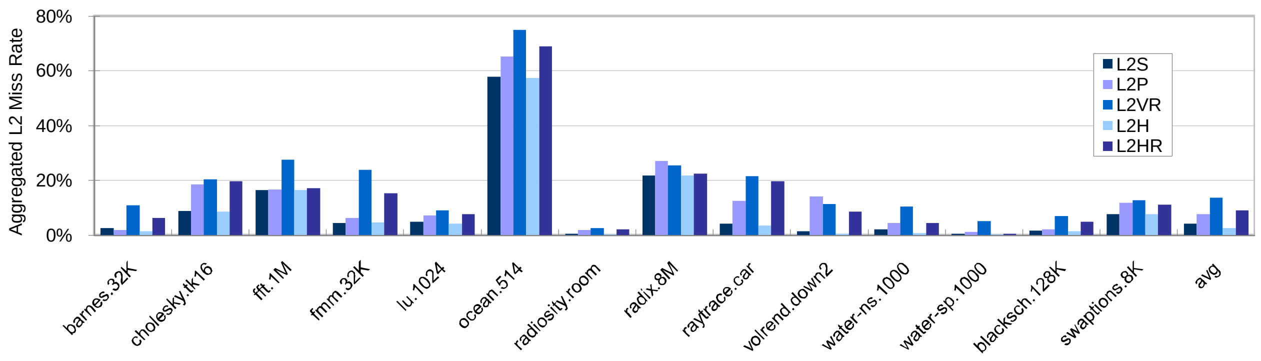


Figure 21: Aggregated L2 cache miss rate of benchmarks with the median input set.

performance by nearly 9% over **L2VR** and **L2P**. It marks 19% performance improvement over **L2S**. These results demonstrate that **L2H** is complementary to other hardware-based optimization techniques. When hints are used to control both private data placement and shared data replication, **L2HR** removes unnecessary replications by allocating private data locally in L2 cache. This helps reduce the cache pressure significantly, resulting in fewer off-chip accesses. On the other hand, **L2VR** cuts the remote access latency of **L2H** by duplicating shared data in local L2 cache. **L2HR** entertains the advantages of both private cache and shared cache schemes.

5.3.3 Cache Miss Rate Comparison

Figure 21 provides more insights regarding to the performance difference in Figure 20(b). It shows the ratio of total number of off-chip reads to the total number of L2 accesses. In general, **L2P** incurs more off-chip accesses than **L2S** due to conflict misses caused by smaller effective L2 cache size. In case of *cholesky*, *ocean*, *raytrace* and *volrend*, **L2S** performs better than **L2P** because the gain of more on-chip accesses in **L2S** is large enough to compensate for the loss in longer remote access latency. However, the margin is very small for the rest of the benchmarks. **L2VR** increases the cache pressure even more than **L2P** as it has to duplicate private data that are blindly distributed in **L2S**. This explains why **L2VR** has worse performance than **L2P** in Figure 20(b). **L2H** has a comparable off-chip access rate as **L2S** since it is basically the shared cache with improved data locality. **L2HR** mitigates the

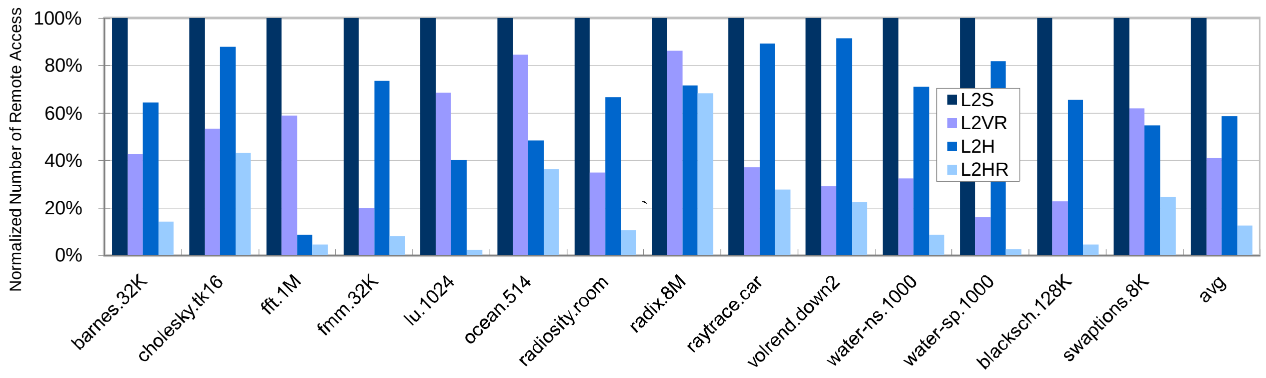


Figure 22: Number of remote L2 data requests normalized to **L2S** for 3 shared cache variants **L2VR**, **L2H** and **L2HR**.

cache pressure of **L2VR** by correctly placing the private data. It effectively improves the L2 cache miss rate of **L2VR**, binging it close to that of **L2S**. However, unlike **L2P**, **L2HR** writes back the modified cache line directly to its home node. This eliminates the need for expensive three-way cache-to-cache transfer of modified data when requested later by other threads.

5.3.4 Network Traffic

The other determinate factor of the distributed shared cache performance is the L2 cache access latency which is affected by the number of remote accesses. Figure 22 shows the number of remote data requests for **L2S** and its variants **L2VR**, **L2H** and **L2HR**, normalized to that of **L2S**. These four schemes have a similar number of L1 misses. Therefore, the more remote L2 data requests there are, the longer the average L2 access latency. Figure 22 shows that **L2VR** effectively removes nearly 60% of the remote accesses of **L2S** by duplicating clean L1 victims in the local L2 cache slice. However, the remote access reduction comes at the sacrifice of the decreased L2 cache hit rate as shown in Figure 21. **L2H** has on average 40% less remote accesses than **L2S** without any compromise in the L2 cache hit rate, since it essentially rearranges the data distribution of **L2S**. Because **L2H** only optimizes the private data while **L2VR** replicates any clean data blocks, it has less remote access reduction than

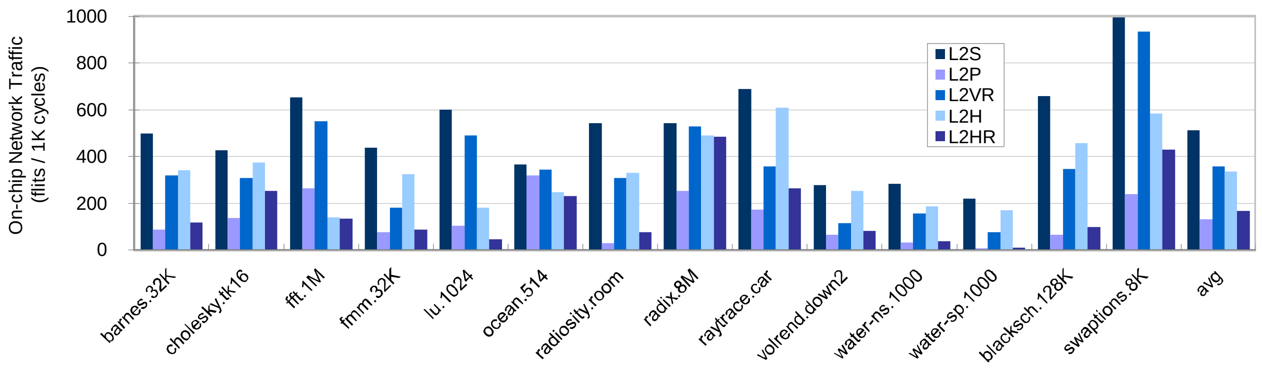


Figure 23: On-chip network traffic of benchmarks with median input set.

L2VR. Surprisingly, **L2HR** eliminates over 85% the remote accesses of **L2S** on average. There are two reasons for this result. First, **L2HR** reduces the number of remote accesses by correctly distributing data in the local L2 cache in the first place. This also eliminates the need for replicating those data as in **L2VR**. Second, a smaller amount of victim replication leads to lower cache pressure, which in turn preserves more replicas of shared data in the local L2 cache.

Besides remote data requests, there are a lot of other network traffic such as coherence messages and memory requests. Figure 23 shows the on-chip network traffic.It is assumed the flit width is 8 bytes. Thus coherence packet takes 1 flit while data packet needs 9 flits. As shown in Figure 23, **L2S** has 3 times more network traffic than **L2P** on average because of the remote accesses. **L2VR** and **L2H** both achieve around 30% traffic reduction by satisfying more data requests in the local L2 cache slice. Finally, **L2HR** reduces the network traffic of the shared cache scheme to the level close to **L2P**.

5.4 SUMMARY

In this chapter, the software-oriented cache management approach is applied to improve the performance of latency-oriented multithreaded programs. The following contributions are made through the study of the proposed scheme:

- A classification of the memory access patterns is carried out for latency-oriented multi-threaded programs. Based on that, an efficient hint-guided data placement and replication scheme is proposed. The scheme is substantially deviated from existing hardware-based schemes. The scheme is orthogonal to the existing hardware schemes. As a result, it can work together with them for even higher performance.
- A novel memory access pattern recognition algorithm is proposed based on the K-means clustering method. The results show that the algorithm works well in recognizing those commonly seen access patterns for dynamically allocated memory areas. The recognized patterns are independent on program parameters. They are general enough to be used for multiple executions with different inputs. This makes the scheme very flexible as the off-line analysis only needs be done once at compile time.
- The proposed scheme is evaluated and compared with the the shared cache scheme, the private cache scheme, and their variants. It is shown that by applying the hints to guide page placement and data replication on the shared L2 cache, it performs significantly better than both the shared cache and the private cache.

6.0 PROFILE-GUIDED DATA REPLICATION FOR MULTITHREADED PROGRAMS

The pattern recognition algorithm introduced in Chapter 5 is effective in capturing the data affinity patterns for latency-oriented multithreaded programs. However, the throughput-oriented multithreaded programs exhibit unique characteristics, which demand a new off-line analysis strategy to improve their data affinity at run time. In this chapter, some insights regarding cache access behavior learned from the profile study of the specjbb server benchmark are presented first. Then the proposed profile-guided data replication scheme for multithreaded programs is introduced in detail followed by the evaluation results. The proposed scheme optimizes the performance of throughput-oriented multithreaded programs through controlled data replication. Even though the focus here is on throughput-oriented multithreaded programs, the proposed scheme can be applied to a broad range of workloads.

6.1 PROGRAM BEHAVIOR ANALYSIS

The CMP architecture is suitable for executing parallel threads. In a CMP, threads can run on separate cores instead of competing for pipeline resources like in a single-core processor. In throughput-oriented multithreaded programs, each thread often independently serves a stream of incoming requests. The last-level cache contains the data for all threads, behaving as a last defense for expensive off-chip memory accesses. An effective use of the last-level cache so that each thread receives good data affinity and low cache interference is critical to the program performance. The problem of placing data in an optimized way in the NUCA last-level cache is as important for throughput-oriented multithreaded programs as other types of programs.

6.1.1 Overview

Throughput-oriented multithreaded workloads are a unique type of programs with growing importance. A representative workload of this type is a server program. In a brief overview, a typical task of a throughput-oriented multithreaded program is to serve requests coming from different sources by returning necessary data or information. Throughput-oriented programs are often programmed in a multithreaded fashion so that multiple requests arriving at the same time can be served simultaneously. Depending on the design, a program can kill an old thread upon the completion of a service and then start a new one for an incoming request. Alternatively, it can command the same thread to keep processing new requests. In both cases, the program essentially continues executing multiple streams of requests until being terminated explicitly. In terms of each individual request, service time is of top priority. But from the perspective of the system, throughput is the most important metric which is often measured as the number of requests serviced during a given time unit. In contrast, latency-oriented multithreaded programs studied in Chapter 5 concern about finishing a given computation task as quickly as possible. Due to the difference in design goals, throughput-oriented multithreaded workloads bear distinct characteristics.

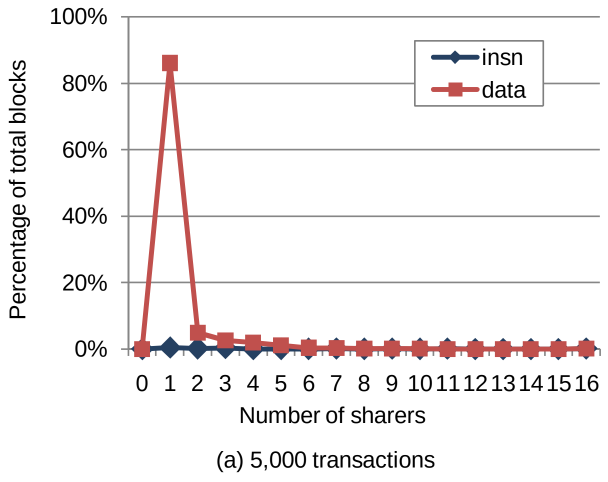
First, the parallel threads of a latency-oriented multithreaded program often work in a cooperative way to tackle a large computation task together. The cooperation can involve significant amount of data exchange and synchronization. In contrast, a throughput-oriented multithreaded program usually does not coordinate the execution among peer threads. They work independently to process independent requests.

Second, the cooperation in a latency-oriented multithreaded program naturally leads to data partitioning. That is, a large problem domain is split into subdomains for parallel processing. Thus the corresponding data set is also partitioned. This allows us to express data affinity hints naturally. However, a throughput-oriented multithreaded program does not have such algorithm-level data partitioning behavior. In such a program, a shared database or a set of common files are accessed by all threads. Thus, data sharing can happen through the database when a user requests the related data.

Lastly, the most distinct feature of a throughput-oriented multithreaded program is that it often runs for a much longer period than a latency-oriented multithreaded program. For latency-oriented programs, the lifetime is limited by the amount of computation. But throughput-oriented programs can run infinitely before being terminated explicitly. During its long-lasting execution, the effect of the OS scheduling, virtual memory management, and I/O can come into play. A latency-oriented multithreaded program suffers less from these factors. With those differences, throughput-oriented multithreaded programs exhibit unique data sharing and communication patterns. This nature demands a new strategy when using the software-oriented approach to manage their performance on a distributed shared cache of a CMP.

6.1.2 Distribution of Cache Blocks

To quantitatively understand the behavior of throughput-oriented multithreaded programs, we analyzed the cache access traces of the *specjbb* program collected from profile executions of 5,000 and 10,000 transactions respectively. Figure 24(a) shows how cache blocks accessed by the program are distributed according to the number of sharers during a profile execution of 5,000 transactions. Light red and dark blue curves represent instruction blocks and data blocks respectively. The big spike in the light red curve shows that about 86.1% of all accessed L2 cache blocks are only touched by one thread during the whole profile period. On the other hand, both curves have flat tails extending to the right. That indicates the number of shared cache blocks only accounts for a small portion. More precisely, 12.1% of the total accessed L2 cache blocks are shared data blocks. The number of shared instruction blocks only accounts for 1.4% of the total accessed L2 cache blocks. Figure 24(b) shows the same plot for a profile execution of 10,000 transactions. The results are very similar to Figure 24(a).



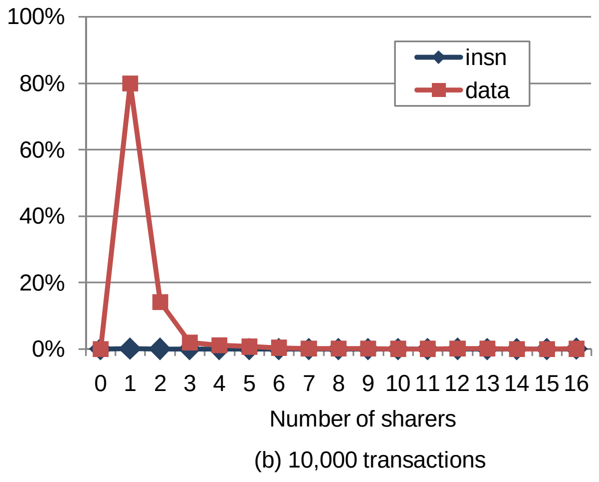


Figure 24: Instruction and data blocks distribution of specjbb 2005 based on the number of sharers of the blocks derived from the profile of (a) 5,000 transactions and (b) 10,000 transactions.

6.1.3 Distribution of Cache Accesses

To further understand the data sharing behavior of the *specjbb*, the distribution of the L2 cache accesses based on the number of sharers from a profile execution is also collected. Figure 25 shows the results. Instruction and data accesses are represented using light red and dark blue curves separately. In comparison with the block distribution in Figure 24(a), the curves in Figure 25(a) have spikes at both ends. At the right end, the number of accesses for instruction and data each accounts for about one third of all L2 cache accesses. The spike at the left end of the light red curve shows that nearly 21% of total L2 cache accesses are private data accesses. Figure 25(b) shows the similar results for a profile execution of 10,000 transactions.

A comparison between Figure 24 and Figure 25 uncovers some insights regarding the access behavior of the *specjbb*. First of all, only less than 1% of all accessed L2 cache blocks are shared by all threads. But they receive about 70% of total L2 cache accesses. This divergence between the distributions of cache blocks and their accesses indicates that data

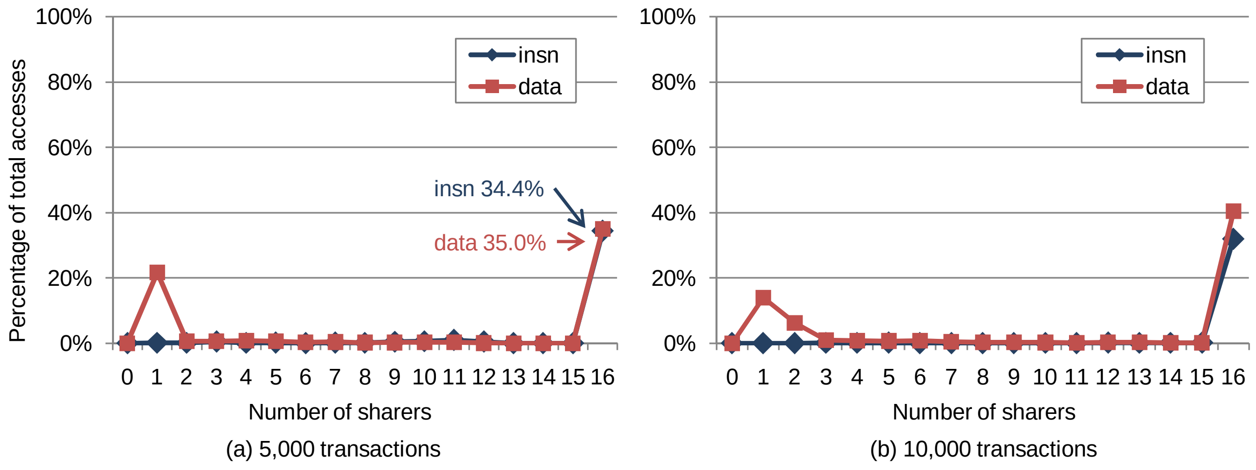


Figure 25: Instruction and data access distribution of specjbb 2005 based on the number of sharers of the blocks derived from the profile of (a) 5,000 transactions and (b) 10,000 transactions.

replication for these highly shared cache blocks can be rewarding. Replication only increases a small margin of the program footprint, but it improves affinity for a significant portion of L2 cache accesses.

Similarly, the divergence for private cache blocks can also be observed. Comparing Figure 24 and Figure 25, it is shown that about 86.5% of all cache blocks are private. But they only receive 21.8% of total L2 cache accesses. This divergence essentially tells that the reuse ratio of private cache blocks is relatively low in comparison to that of these highly shared cache blocks. Because private data have much larger footprint than the highly shared data, uncontrolled replication of private data that are not placed locally can be detrimental when the cache capacity is limited.

In both Figure 24 and Figure 25, the curves in plot (a) and plot (b) show similar shape. That demonstrates the observed cache access behavior of the *specjbb* is largely independent on the length of the profile execution. This observation is worth some more discussion. By comparing plot (a) and plot (b) in both figures, we discover that some cache blocks become

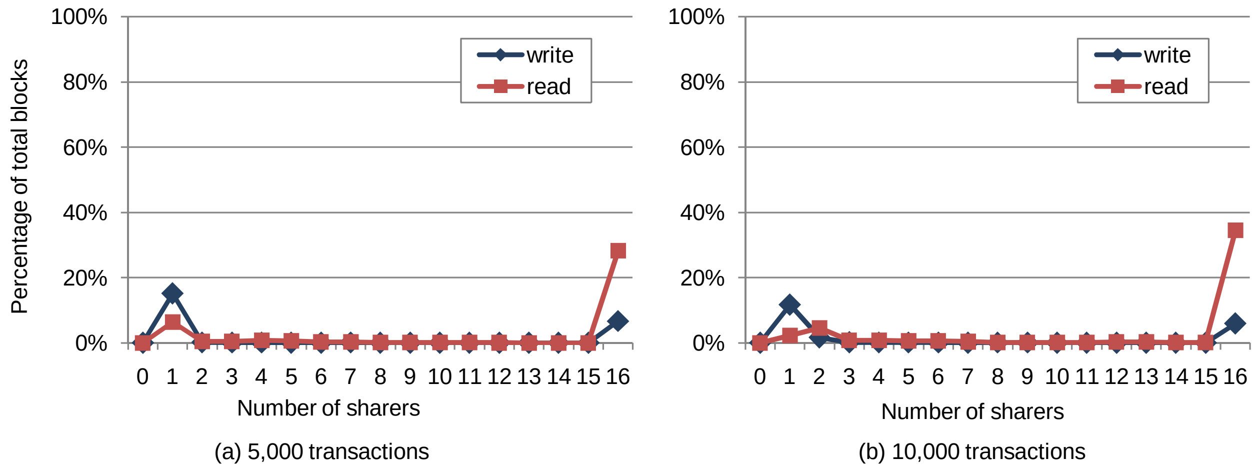


Figure 26: Read/write access distribution of specjbb 2005 based on the number of sharers of the blocks derived from the profile of (a) 5,000 transactions and (b) 10,000 transactions.

shared as the profile execution length increases. This phenomenon is not a coincidence but an artifact from the memory management of the JAVA virtual machine. As the virtual machine constantly reclaims memory area from garbage and maps them to new objects, a private cache block will be used by another thread eventually if the profile execution lasts long enough. These cache blocks are regarded as shared by off-line analysis due to the lack of dynamic information about the remapping. This information is only available at run time. For this reason, a static data placement hint which maps a given virtual address to a fixed location does not work well for this type of workloads. In order to manage data affinity efficiently by using off-line analysis hints, a more flexible strategy is required. Based on this discussion, hint-guided data replication appears a better strategy than the hint-guided data placement scheme.

6.1.4 Distribution of Cache Reads and Writes

Figure 26 further shows the distribution of the L2 cache accesses between read access and write access. Again, the length of the profile execution has little impact on the results since plot (a) and (b) are similar. Furthermore, the figure shows that the cache blocks shared by all 16 threads receive many more reads than writes. Because a write triggers invalidation of data shared by other threads, data replication can increase the cost of this invalidation process. For this reason, a replication scheme would favor data with a high read-to-write ratio. On the other hand, private data have very low read-to-write ratio. A replication scheme should avoid replicating them. Thus these highly shared data are proper candidates for data replication.

6.1.5 Property of Memory Instructions

We also examined data reuse patterns seen by each individual memory instruction and observed an important property for memory instructions: a memory instruction tends to exhibit persistent data access pattern. For instance, if a data block accessed by a load instruction is found to exhibit high reuse, then it is likely that other data blocks referenced by the same load instruction will also exhibit good rescue. The following pseudo-code shows a simple example of a Fibonacci number generator. This example gives us a motivating idea of why a memory instruction can have a persistent data access pattern.

```
data[0] = 0;
data[1] = 1;
for(i = 2; i < N; i++) {
    data[i] = data[i - 1] + data[i - 2];
}
```

In this example, a data array is accessed within a *FOR* loop. Except the first two elements of the data array, each one is read twice for generating the next new number. The newly calculated value is written to the next element in the array. If each array element occupies 4 bytes and a cache block is 64-byte long, a cache block can receive up to 48 accesses in total after it is first accessed. All these reads and writes are initiated by the same statement. In this case, it is evident that a cache block touched by this statement can see more reuse in the near future. Of course, the programs in the real world are much more complex, but they are composed of nothing but loops and all kinds of control flows. As a result, some statements can exhibit a similar property as the one in this Fibonacci example. Since the reuse pattern of such a statement (memory instruction essentially) does not change with program parameters and cache configurations, it can be a good hint for guiding online data replication.

6.2 PROFILE-GUIDED DATA REPLICATION SCHEME

Because of the long execution time of throughput-oriented multithreaded programs and the intervention of the OS memory management module, a static hint-based data placement scheme cannot work well. Luckily, a private cache block and a shared cache block exhibit distinct access patterns. It suggests that controlled replication can be a viable approach. Figure 27 shows the flow diagram for the proposed profile-guided data replication scheme at a very high level. During the profile stage, the L2 cache access trace and the corresponding memory instruction addresses are collected. The analysis routine estimates the reuse pattern for each memory instruction recorded in the trace. Memory instructions with poor reuse behavior are filtered before generating an annotated program binary. Finally, the annotations within memory instructions dictate the data replication decision at run time. In this section, we will first present the trace analysis algorithm for filtering memory instructions. Then we will describe in detail how the reuse pattern information can be utilized at run time to guide data replication in L2 cache.

6.2.1 Trace Analysis Algorithm

To analyze the reuse of memory instructions, an L2 cache access trace is collected during the profile execution. Each trace item contains the memory reference address and the corresponding program counter (PC) address of the instruction, which issues the memory request. A PC address can uniquely identify the corresponding memory instruction. For each memory instruction, a corresponding counter is added for each accessed memory block to track the number of references. Trace items are examined one by one as follows: The counter

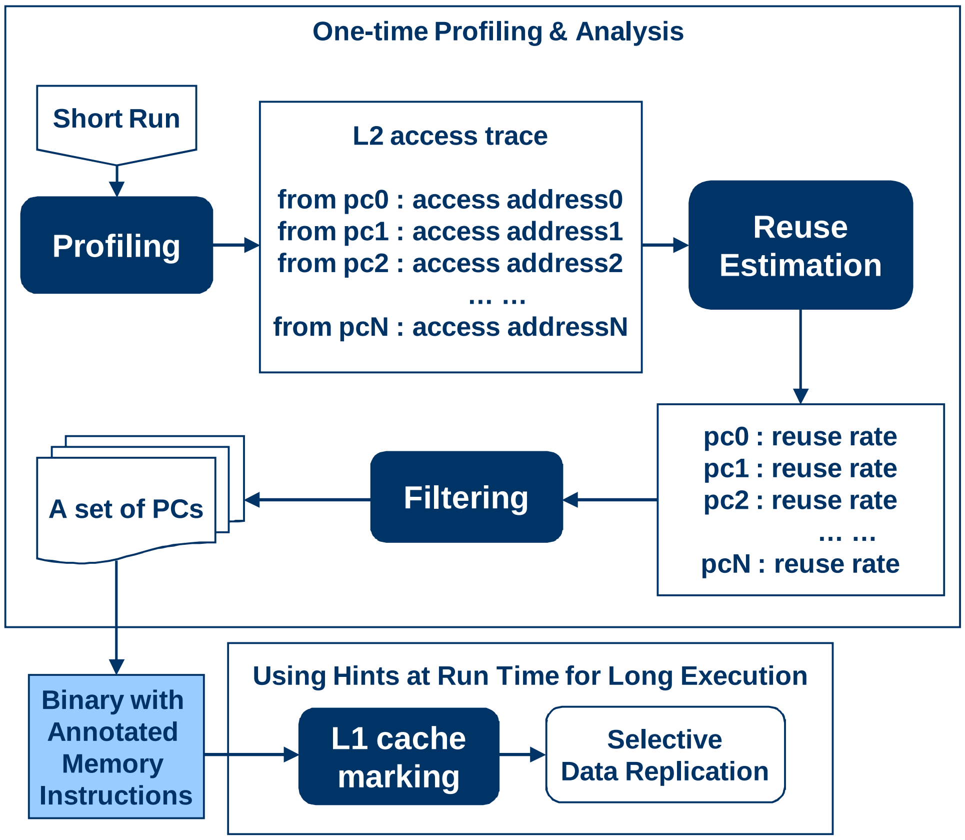

One-time Profiling & Analysis
Short Run
Profiling
L2 access trace
from pc0 : access address0
from pc1 : access address1
from pc2 : access address2
… …
from pcN : access addressN
Reuse Estimation
pc0 : reuse rate
pc1 : reuse rate
pc2 : reuse rate
… …
pcN : reuse rate
Filtering
A set of PCs
Binary with Annotated Memory Instructions
Using Hints at Run Time for Long Execution
L1 cache marking
Selective Data Replication

Figure 27: Block diagram for the one-time profile and trace analysis process (upper box) and the run-time data replication control (lower box).

array of the memory instruction is first selected using the PC address. Then the counter corresponding to the referenced block address is looked. If such a counter is not found, a new one is created and added into the array. Otherwise the counter value is incremented. If the PC address has not been seen before, an empty array is created for it. At the end of this process, these counters represent the number of accesses received by different cache blocks for a specific load instruction.

The following pseudo code demonstrates this process. The elements of the *PC_list* are indexed by the PC address of a memory instruction. Each of them points to a *BLK_list*, which counts the number of access for each data block. Access counts can be tracked at different granularity. Cache block granularity provides the most precise information but coarser granularity requires less computation power and storage. In this study, cache block granularity is adopted to gain maximum accuracy. Only load instructions are considered in the algorithm.

```
/* trace processing */
while trace is not empty:
    PC = pc address of the memory instruction
    BLK = blk address of the memory reference

    if PC in PC_list:
        BLK_list = PC_list[PC]
        if BLK in BLK_list:
            BLK_list[BLK] = BLK_list[BLK] + 1
        else:
            BLK_list[BLK] = 0
    else:
        add PC to PC_list
```

After executing this algorithm with a collected trace, the corresponding *BLK_list* of each memory instruction contains the access counts for all data blocks accessed by this memory instruction.

The next step is to identify memory instructions that can bring in data blocks with many potential reuses. The underlying assumption here is that each data block requires at least one memory access to load the value. Thus a memory instruction's total number of memory accesses is equal to the summation of the counter values in *BLK_list* array while the number of reuses is calculated as the summation of each counter value in *BLK_list* array minus one. Then the reuse ratio for a memory instruction is simply the number of reuses divided by the total number of memory accesses. In our algorithm, a memory instruction is considered as having good temporal reuse if the reuse ratio is larger than a pre-defined threshold. A small threshold value passes a memory instruction with a low reuse count, resulting in more qualified memory instructions than a larger threshold value. The following pseudo code demonstrates this algorithm.

```
/* PC filtering based on reuse rate */
for each PC in PC_list:
    BLK_list = PC_list[PC]
    reuse_count = 0
    access_count = 0
    for each BLK in BLK_list:
        access_count = access_count + BLK_list[BLK]
        reuse_count = reuse_count + BLK_list[BLK] - 1

    /* found a PC with high reuse rate */
    if reuse_count / access_count > Threshold:
        print PC
```

This filtering method essentially differentiates two types of memory instructions. One type of memory instructions touch many memory blocks. But the access count for each block is low. This can lead to a low reuse ratio based on our algorithm. The other type of memory instructions access a memory block a lot after loading it into the cache. As a result, the *reuse_count* is very close to the *access_count*, leading to a high reuse ratio.

6.2.2 Profile-Guided Data Replication

Another challenge of using a software-oriented approach to optimize the performance of throughput-oriented multithreaded programs is how to convey the static data reuse hints to accessed data blocks dynamically at run time. The off-line analysis is performed at the time of program compilation. Using the algorithm introduced in Section 6.2.1, the compiler can identify a list of memory instructions that access data with good reuse property from a profile trace of L2 cache accesses. At the code generation phase, these memory instructions are marked with hints to differentiate them from other normal memory instructions. All memory instructions access the data in the same way as before. An L1 miss triggers an L2 cache access, which brings back the data. But if a memory instruction is marked, its accessed L1 cache block also has to be marked to indicate that it may experience a lot of reuse in the L2 cache. At this point, the hint regarding a memory instruction is transferred to an L1 cache block. This cache block is expected to receive some reuses in the future.

When an L1 cache block is evicted, some extra actions are performed. The cache block is first checked to see if it is marked. If the flag is not set, it is retired by following the normal cache coherence routine. Otherwise, the cache block is deemed as a candidate for replication in the local L2 cache. To find a place in the local L2 cache, the scheme searches for an invalid cache block, an unused cache block or a replicated cache block in order, same as the victim replication scheme [111]. If no such a L2 cache block is found, the L1 cache block has to be evicted. Because timely fetching of instructions is critical to the pipeline performance, an instruction cache block is always a candidate for replication in the local L2 cache. The profile study shows that instructions have a very small footprint. So the replication of all instruction cache blocks has little impact on the cache pressure.

The number of cache blocks that exhibit good reuse can grow when the execution length increases. The cache blocks accessed also tend to change under different execution environment. As a consequence, there is no way to directly supply accurate replication hints in terms of data blocks that are expected to be accessed at run time. However, the key memory instructions that always access data blocks with good reuse property stay largely unchanged. This observation makes the profile-guided data replication possible and is the major contribution of this work. By attaching the reuse information to memory instructions and then transferring them to access cache blocks at run time, the hints become decoupled from program parameters and cache configurations.

This profile-guided data replication scheme only allows data accessed by specified memory instructions to be replicated. These data are expected to have good reuse property. Other data are considered to have poor temporal locality, thus not replicated. Avoiding the replication of data with poor locality limits the interference from reckless replication. This can effectively relieve cache pressure caused by excessive data replication. On the other hand, replication of data with potential reuse effectively improves access latency of these frequently accessed cache blocks.

6.2.3 Implementation Issues

Architectural change is required to support the proposed profile-guided data replication scheme. Since most of the trace analysis and decision making process in this proposed scheme are performed off-line by software, there is no need for hardware logic and storage to track program behavior at run time. The added hardware complexity is minimized. The cache structure is similar to a traditional distributed shared cache. Each L1 data cache block needs one extra flag bit to indicate if the data block is replicable at the time of eviction. This flag is set based on the annotation comes with a memory instruction. Accordingly, the ISA has to provide a set of modified memory instructions with *hint bit* added. The compiler turns on or off the bit based on the decision made from the profile analysis. In case there is no spare bit available in an existing ISA design, hints can be carried explicitly with the program binary in a form of a list of PC addresses. Depending on the size of the provided hardware storage, however, the length of the list can be constrained. In addition, a hardware logic for matching the PC address of the current memory instruction with the hint list is required. For a long list, an efficient matching mechanism should be implemented. In this work, the annotated ISA approach is assumed.

The replication management and the corresponding cache coherence changes are similar to the victim replication scheme. No more complexity is added. The area overhead of the added *hint bit* in the L1 data cache is very limited. Attaching one bit to every 64-byte cache line only incurs less than 0.2% area budget for an L1 data cache. There is no change for the L1 instruction cache. The time it takes to make the data replication decision is overlapped with the latency for fetching the new data block. Thus it is off the critical path. If no replication is allowed for an evicted data block, it can take extra time for the directory to get notified. But the impact is minimum because the notification is asynchronous to the pipeline execution.

6.3 EVALUATION RESULTS

To evaluate the proposed scheme, *specjbb 2005* [95] and *apache* [4] are chosen as the server workloads. In addition, we also customized two server kernel programs: btree and cheetah [58, 92]. Btree simulates database activities by performing lookup, insert, and delete operations on an existing B-Tree structure. Cheetah simulates the behavior of a static web server with a modified network component to ease the simulation setup. Besides, selected programs from SPLASH-2 [86] and PARSEC [76] are also evaluated. Four major cache schemes are simulated and compared: the shared cache scheme (**L2S**), the private cache scheme (**L2P**), the victim replication scheme (**L2VR**), and the proposed hint-guided data replication scheme (**L2HR**).

6.3.1 Profiling Analysis Threshold

The threshold value in the trace analysis algorithm plays an important role in determining the amount of data replication allowed for a given program. A small threshold value passes memory instructions with low reuse ratio as hints, leading to more replication at run time. One extreme is to use 0 as the threshold value, which generates hints for all memory instructions. It essentially enables replication for all accessed data, behaving exactly as the victim replication scheme. A large threshold value does the opposite. A threshold value of 1.0 results in no hints at all, directing the hint-guided data replication scheme to fall back to the shared cache scheme. It is crucial to select a proper threshold value based on a program's characteristics. In this work, six threshold values are examined: 0, 0.2, 0.4, 0.6, 0.8 and 1.0. The threshold that results in the best performance for the testing execution is used to generate the hints.

Table 4 shows the number of static memory instructions captured as hints by our off-line analysis algorithm. As expected, the number of captured memory instructions decreases with a larger threshold value. Threshold 0 essentially shows the total number of memory instructions seen by the L2 cache. Figure 28 shows the normalized execution time of programs with the guidance of data replication hints generated from the six threshold values. The

Threshold	0	0.2	0.4	0.6	0.8
specjbb	3485	3125	2996	2841	2508
apache	3313	2948	2771	2587	2307
btree	2009	1901	1849	1751	1518
cheetah	1279	1251	1203	1134	994
barnes	1072	1001	980	955	838
fmm	1594	1502	1452	1347	1176
radiosity	1090	1007	971	931	850
raytrace	1263	1152	1137	1112	995
volrend	800	774	761	738	683
blackscholes	2596	2541	2503	2414	2087
swaptions	2868	2720	2657	2496	2164

Table 4: Number of static memory instructions captured by different threshold values.

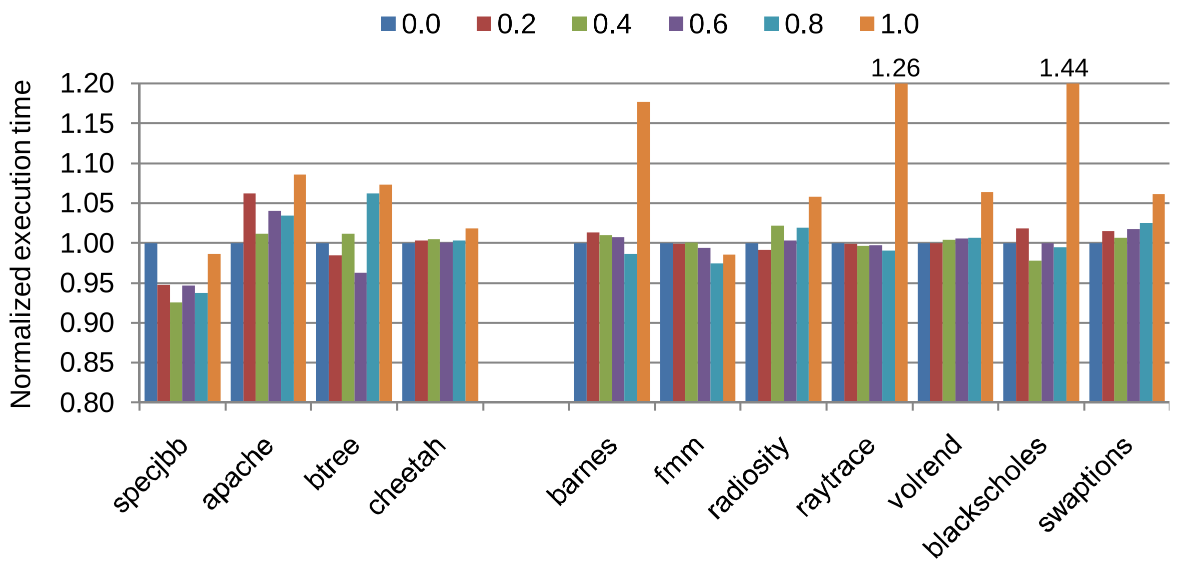


Figure 28: Impact of threshold value on program execution time.

Program	specjbb	apache	btree	cheetah	barnes	fmm
Threshold	0.4	0	0.6	0	0.8	0.8
Program	radiosity	raytrace	volrend	blackscholes	swaptions	
Threshold	0.2	0.8	0	0.4	0	

Table 5: The optimal threshold value derived from test executions is given for each program. A high threshold value filters out more memory instructions, resulting in less data replication. A threshold value of 0 represents the victim replication scheme.

examined programs exhibit different preference on the threshold values. For instance, *specjbb* performs best with a threshold value of 0.4 while *apache* prefers a threshold value of 0. Table 5 lists the optimal threshold values for all evaluated programs. From here on, results reported for hint-guided data replication scheme are evaluated using the optimal threshold values unless otherwise noted.

6.3.2 Reuse Comparison

After obtaining the optimal threshold values, we evaluate programs with much longer execution time. The first thing we are interested in is how well the hint-guided replication scheme controls the data replication amount. Figure 29 compares **L2VR** and **L2HR** in terms of their replication effectiveness. The bars in the figure represent the percentage of the total number of L2 cache accesses that get hit in local L2 caches because of data replication. The figure shows that only for few programs **L2HR** has slightly less number of local L2 cache hits than **L2VR** since it constrains data replication. Overall, **L2VR** and **L2HR** have very similar amount of local L2 cache hits that are served by replicated data. This figure demonstrates that **L2HR** still preserves most of the performance benefits of **L2VR**, even though it limits data replication for some L1 cache evictions. Figure 30 shows the number of data replication instances that occurred in **L2HR** relative to **L2VR**. The results show that programs with high threshold values, such as *specjbb*, *btree*, and *raytrace*, have signifi-

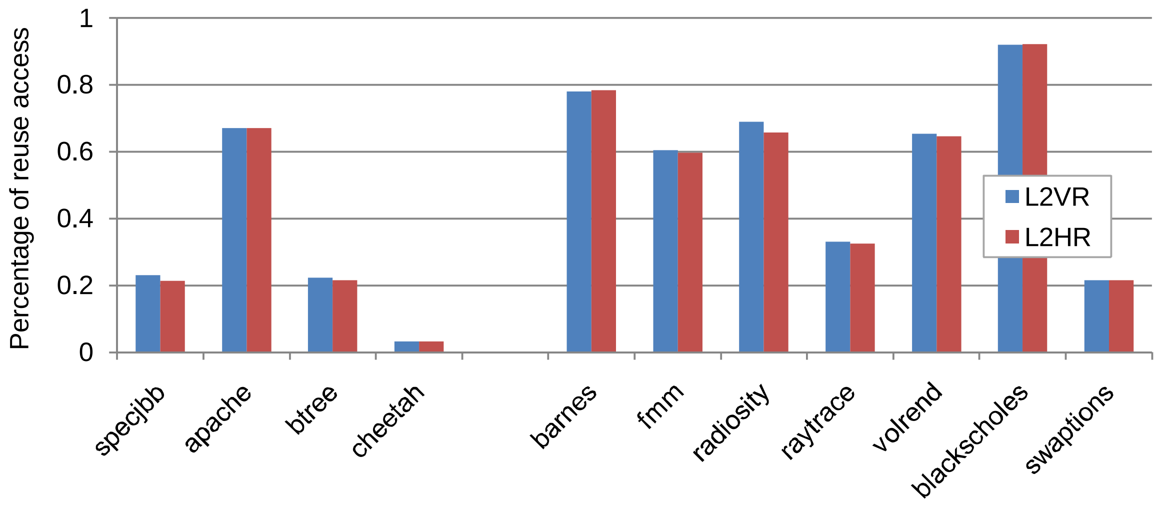


Figure 29: A comparison of the ratios of the total number of L2 cache accesses that find data replicated in local cache slices for **L2VR** and **L2HR**.

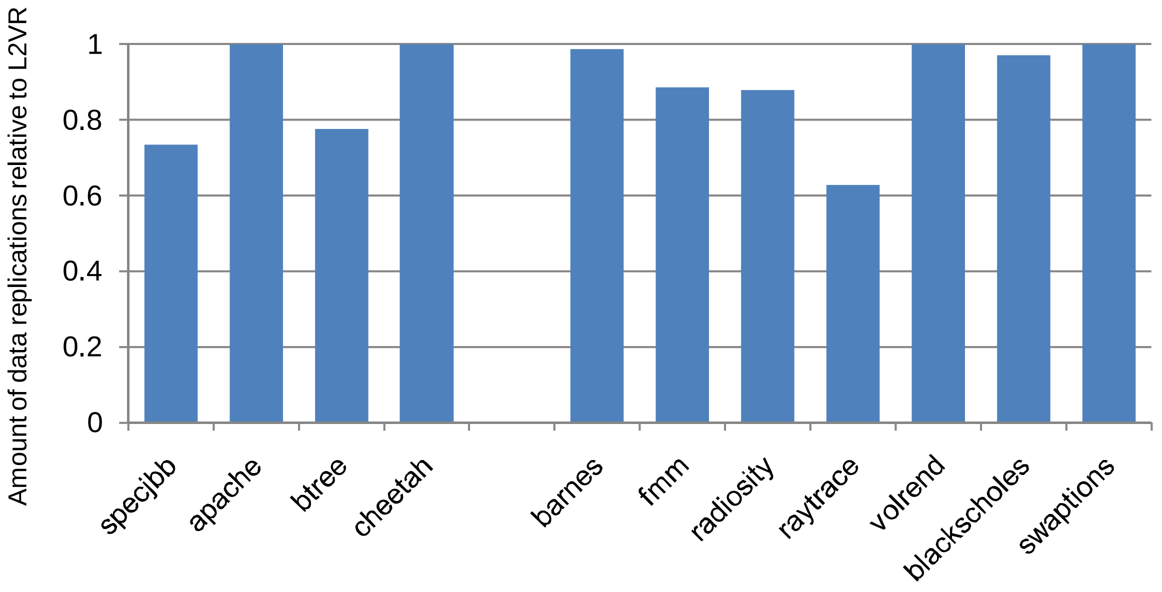


Figure 30: Number of data replication instances in **L2HR** relative to **L2VR**.

cantly less data replication than **L2VR**. It illustrates a nice property of the proposed data replication scheme: by adopting a carefully selected threshold value, useless data replication can be largely suppressed while not affecting the reuse ratio in local L2 caches.

Figure 31 uncovers the other side of the story. It compares the aggregated L2 cache miss

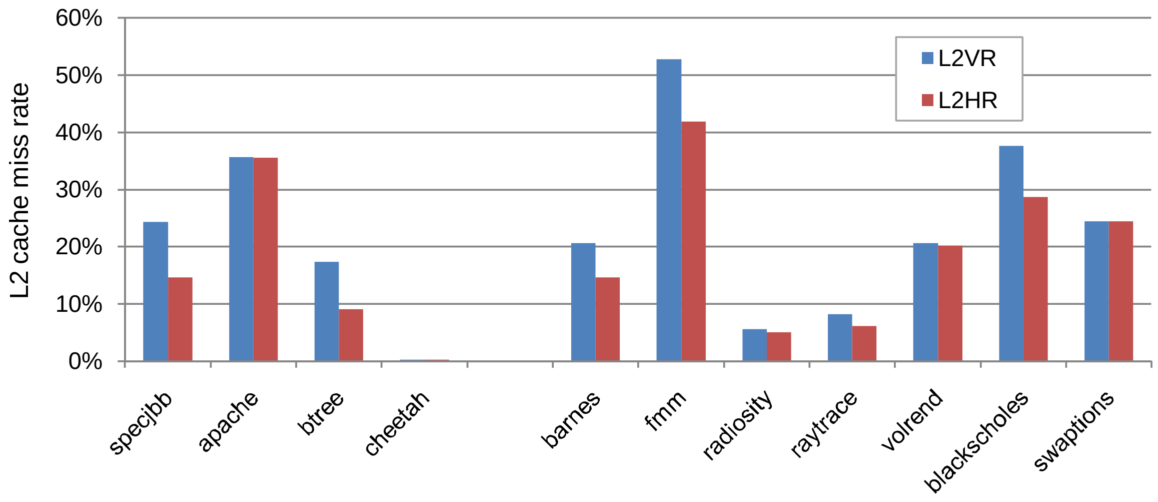


Figure 31: A comparison of the aggregated L2 cache miss ratio for **L2VR** and **L2HR**.

rate between **L2VR** and **L2HR**. Compared to **L2VR**, **L2HR** reduces cache miss rate by 9.7% and 10.9% for *specjbb* and *fmm* respectively. For other programs such as *btree*, *barnes*, and *blackscholes*, **L2HR** also achieves considerable cache miss rate reduction. As expected, these programs all have high threshold values shown in Table 5. A high threshold value prohibits many data replication instances that are allowed in **L2VR**, resulting in less contention in the L2 cache. Figure 29 and Figure 31 together suggest that the data replication hints derived from off-line analysis are effective in removing fruitless data replication instances while keeping data with potential reuse in the L2 cache.

6.3.3 Performance

Lastly, we examine the performance of the proposed **L2HR** scheme with comparison to **L2S**, **L2P**, and **L2VR**. We use execution time as the metric to evaluate performance. For latency-oriented multithreaded programs, execution time of the parallel section is measured. For throughput-oriented multithreaded programs, the execution time is measured for a given number of completed transactions. As a result, the measured execution time is inversely proportional to a program's throughput. Figure 32 shows the results, which are normalized to the shared cache scheme. Overall, **L2P** performs considerably worse than **L2S**. Especially, for *btree*, *cheetah*, *raytrace*, and *swaptions*, **L2P** performs 38.0%, 71.1%, 483.8%, and 52.8%

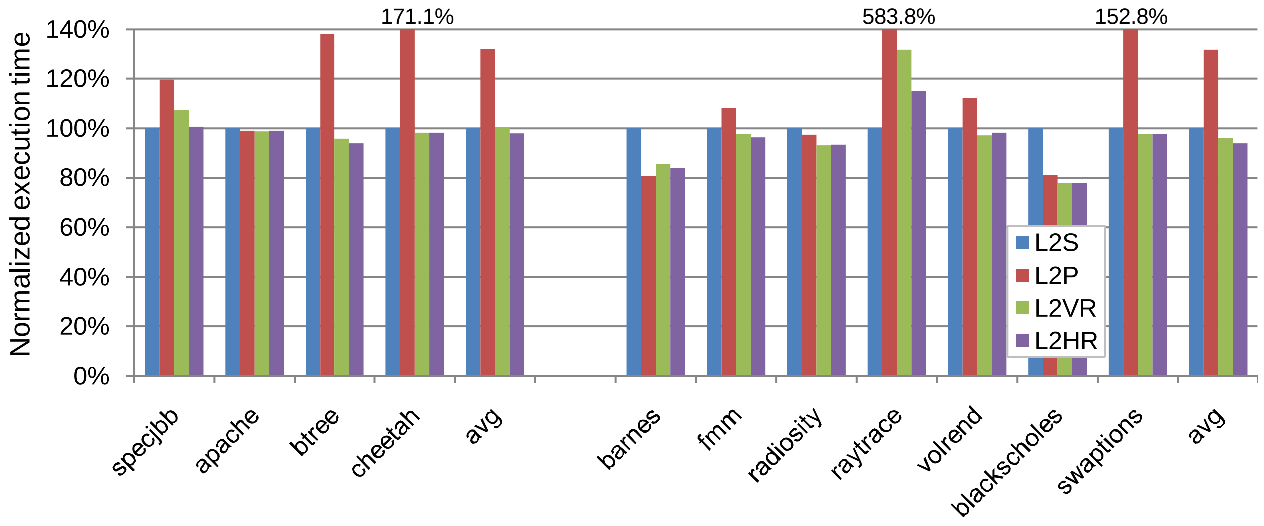


Figure 32: Normalized execution time of **L2S**, **L2P**, **L2VR**, and **L2HR** schemes.

worse than **L2S**. The performance degradation of **L2P** is mainly caused by excessive L2 cache misses. For instance, *cheetah* misses almost all local L2 cache accesses with **L2P** scheme. This is because serving a HTTP file request exhibits little data locality. Caching data in local private L2 cache does not help in this case. However, caching data in shared L2 cache can provide potential reuse for the same requests from other threads. **L2VR** tracks the best performance between **L2S** and **L2P** very well. When **L2P** shows advantage over **L2S**, **L2VR** starts to perform better than **L2S** by shrinking the number of remote accesses. For example, **L2VR** has similar performance as **L2P** for *barnes* and *blackscholes*, achieving 14.5% and 22.1% better performance than **L2S**.

Our proposed **L2HR** scheme approaches the best of all these three schemes. Most of the time, **L2HR** tracks the performance of **L2VR** closely. In case **L2VR** fails to adapt to a large working set of a program, **L2HR** can safely fall back to **L2S**. For example, **L2VR** performs worse than **L2S** for *specjbb*, in which case **L2HR** can constrain the data replication and approach the performance of **L2S**.

To examine how replication level affects a program's performance, we introduce probabilistic data replication. Instead of indiscriminately replicating all L1 cache victims as in **L2VR**, probabilistic data replication only keeps L1 cache evictions in local L2 cache with a pre-defined probability. We use **L2VRxx** to indicate that data replication is allowed in

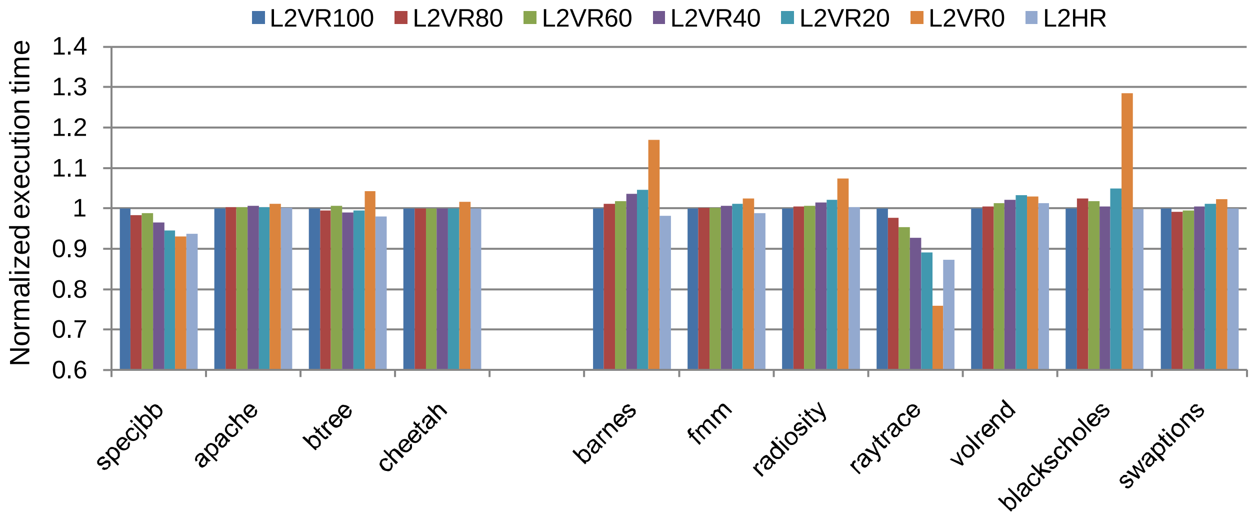


Figure 33: Normalized execution time of variants of **L2VR** and **L2HR**.

this scheme with a probability of **xx%**. Thus **L2VR0** is equivalent to **L2S** while **L2VR100** represents **L2VR**. We examine probabilities ranging from 0 to 100 with an interval of 20.

Figure 33 compares their performance with **L2HR**. We can make several interesting observations from the figure. Firstly, some programs such as *specjbb*, *barnes*, *radiosity*, *raytrace*, and *blackscholes* are sensitive to replication probabilities. Others do not respond to replication that much. Secondly, *btree*, *barnes*, *radiosity*, *raytrace*, and *blackscholes* all show big performance difference between **L2VR20** and **L2VR0**. This indicates that these programs have some frequently accessed data, which benefit much from replication. Lastly, we can observe that **L2HR** approaches the best probabilistic data replication scheme for most of the programs. The results clearly demonstrate that **L2HR** through off-line trace analysis can uncover critical data replication information and use them as hints at run time to improve data replication efficiency.

6.4 SUMMARY

In this chapter, the cache access behavior of throughput-oriented programs is studied. Using *specjbb* as a case study, we show that server programs exhibit divergent access behaviors for privately owned data and highly shared data in the L2 cache. Motivated by this observation, then we propose a hint-guided data replication scheme. The new scheme relies on the off-line trace analysis to generate replication hints on a per-memory-instruction basis. The evaluation results show that most of the selected latency-oriented programs perform well without data replication control for the given architecture configuration. The victim replication scheme works well. For the throughput-oriented programs, the proposed controlled data replication scheme improves performance moderately compared to the victim replication scheme.

Due to the dynamic nature of throughput-oriented programs, using profile-derived information to guide online decision seems a big challenge. There is no previous work that studies the profile-driven approach to manage shared L2 cache for improved performance for throughput-oriented programs. This work contributes to the literature by attempting to explore a promising method in this direction.

7.0 CONCLUSIONS AND FUTURE WORK

7.1 SUMMARY

This thesis proposes and studies a software-oriented shared cache management approach for CMPs. The new approach proposes to augment two extra fields to TLB and page table entries so that the address of a last-level cache access can be translated in a controlled way. The cache access address is formed by replacing designated bits from the original physical address by the values from the augmented fields. The main memory access still uses the original physical address. Thus it is decoupled from the last-level cache access. The augmented fields introduce moderate increase in the page table storage, but this minor modification to the hardware wins many opportunities. Instead of distributing data blindly like the classic shared cache scheme, the proposed approach allows software components to manipulate data placement in the last-level cache by supplying proper values for the augmented fields in page table entries. Compared to hardware-based cache schemes that manage data at cache block granularity, the proposed approach raises the data placement granularity to page size. By sacrificing control granularity to control flexibility, the proposed approach brings many optimization opportunities for shared last-level caches of CMPs. For instance, cautious data placement can have a profound impact on a program's performance, quality of service, power consumption, reliability, fault tolerance, and so on. This thesis studies how to utilize this flexible data placement mechanism to improve program performance by extracting critical hint information through off-line trace analysis.

Since the software-oriented shared cache management approach offloads majority of the management task to the off-line trace analysis, the decision making process is no longer restricted by the response time and hardware resources. Sophisticated trace analysis algorithms can be performed to extract the best available information. Especially, it allows different strategies to be applied based on the characteristics of various programs. This study is broadly split into three parts, each of which mainly targets one of three most important types of modern workloads.

For single-threaded programs, the proposed static 2D page coloring scheme derives an optimal page placement schema through off-line analysis of the trade-off between remote cache access latency and the cache miss rate for a collected L2 cache access trace. The generated hints are then used to guide the program execution with the same input set. The results of this oracle scheme indicate that it effectively removes unnecessary remote cache accesses and mitigates cache contention, leading to superior performance for single-threaded programs running on CMPs. Due to the ideal setting, it provides an upper bound for optimization of this kind. A more practical solution, named dynamic 2D page coloring scheme, is proposed. Following the same philosophy, it utilizes online information to make best data placement decisions. However, the initial page placement decisions can be sub-optimal due to the lack of accurate information regarding the future data usage. To recover the impact caused by improper initial page placement decisions, a page migration scheme is also studied. It is used in conjunction with the dynamic 2D page coloring scheme when enabled. The evaluation results show that the dynamic 2D page coloring scheme approaches the performance of the static scheme very well. The additional speedup achieved by the page migration scheme is moderate, but it rescues the performance in case the initial page coloring scheme fails to make proper decisions.

The second type of workloads is latency-oriented multithreaded programs. The fundamental difference between this type of workloads and single-threaded programs is represented by the data sharing behavior among parallel threads in latency-oriented multithreaded programs. A block of shared data can be possessed by multiple threads simultaneously. Moving a block of shared data closer to one of its sharers can make it apart from other sharers, effectively leading to worse performance for the majority of sharers. On the other side, placing a block of shared data in the middle of all sharers may improve overall fairness. However, every sharer can end up being worse off than the optimal access latency. As a result, determining the optimal location for a block of shared data is not a trivial task. This thesis introduces a pattern recognition algorithm based on the K-means clustering method to identify commonly seen cache access patterns. Generated hints are used to guide data placement for private data and data replication for highly shared data. Through data replication, the described dilemma for shared data placement is largely avoided. The increased cache pressure incurred by data replication is mitigated nicely through two ways: First, data affinity hints direct private data placement to their owners' local cache slices. The amount of data replication for private data can be reduced significantly, if they are correctly placed. Second, data replication is mostly limited to shared data, which tends to have small footprint. So replication only increases the cache capacity demand moderately. The evaluation results show that the pattern recognition algorithm is effective in identifying data distribution patterns. The derived hints guide data placement and replication properly, improving program performance substantially.

The last type of workloads is throughput-oriented multithreaded programs. Due to their unique characteristics, the hint-guided data placement proven to be effective for latency-oriented programs does not work well. Indeed, throughput-oriented programs are very difficult to optimize by using off-line analysis hints, because they tend to have many non-deterministic activities. It is not possible to capture these dynamic characteristics through static off-line analysis. The dynamics of thread creation and termination also makes static data placement futile. The solution proposed in this thesis tries to recognize data that can receive potential reuses in the future. Through controlled replication of those data, data affinity is improved while cache contention is confined. One important observation that lays the foundation for the proposed controlled replication scheme is: the data reuse property of a memory instruction is determined by the algorithm design of a program. If a block of data accessed by a memory instruction shows good temporal locality, then other data blocks accessed by the same memory instruction can exhibit the same reuse behavior. Memory instructions that access data with high reuse rate can be identified through off-line analysis of a profile trace. As a result, the dynamic information regarding data reuse at run-time can be conveyed using static hints. Moreover, this scheme is also applicable to other multithreaded programs. The experimental results show that the evaluated server programs achieve meaningful performance improvement over the *victim replication* scheme. However, the improvement for other latency-oriented multithreaded programs was limited with our current implementation of the idea.

7.2 CONCLUSIONS

This work proposes a software-oriented shared cache management approach as an alternative to existing hardware-based cache management schemes. Thanks to the flexibility of the software-oriented data placement mechanism, various off-line analysis schemes can be integrated in a single framework to target different types of workloads. In this work, three cache management schemes are proposed and studied. The following conclusions can be drawn from the qualitative analysis and the experimental results:

- Cautious data placement is very important to the performance of single-threaded programs that run on non-uniform latency shared caches. Through careful data placement, a significant portion of cache contention in the traditional shared cache scheme can be removed. This results in a huge saving of the bus bandwidth to off-chip main memory. Furthermore, location-aware data aggregation is very effective in cutting down the average remote cache access latency. These two optimizations together achieve significant performance boost for single-threaded programs. This shows that off-line analysis has advantages over online approaches, despite it lacks dynamic information.
- Page-level data migration is also studied for single-threaded programs. This dynamic scheme helps correct improper data placement decisions when more page usage information is available. But the performance improvement is limited because of its large migration overhead. We find that migration decisions must be made carefully due to the same reason.
- Analysis shows that latency-oriented multithreaded programs exhibit many recognizable cache access patterns, which are intrinsic properties of program designs. These patterns can be utilized to guide online data placement and data replication. The proposed pattern recognition algorithm is effective in identifying these patterns. The derived hints successfully improve programs' data affinity, achieving good performance improvement.

The hint-guided data placement scheme achieves comparable or even better performance than other hardware-based cache management schemes including the victim replication scheme and the dynamic spill and receive scheme.

- Throughput-oriented programs are very difficult to optimize by using hints generated off-line from cache access traces. Their dynamics and non-deterministic activities are hard to characterize through trace analysis. But reuse pattern of data fetched by the same instruction tend to be similar. As a result, we characterize reuse behavior of memory instructions and use these information as hints to guide online data replication. The proposed hint-guided data replication scheme shows promising results for server programs. Overall, it achieves comparable performance improvement as other hardware-based schemes.
- This thesis work shows that the proposed software-oriented shared cache management approach is promising. It enables off-line analysis to guide online data placement and data replication. In general, its achieved performance improvement is competitive with and sometimes better than other hardware-based cache schemes. However, the hardware complexity of the proposed software-oriented approach is significantly lower. This provides much benefit in terms of power consumption, reliability, and design efficiency.

However, the software-oriented approach has its limitations as compared to hardware-based schemes:

- The effectiveness of off-line trace analysis is very sensitive to the OS scheduling. While the hints can accurately summarize the data access patterns for a particular program, they may not apply to the actual condition where multiple programs are co-scheduled.
- The proposed static 2D page coloring scheme may not be practical in many situations. The generated data placement hints have to be used for the execution with the same data input. The dynamic 2D page coloring is a more realistic scheme.
- While the proposed hint analysis algorithm for latency-oriented multithreaded programs is largely independent on input size and many architectural parameters, it requires fixing the number threads for some types of patterns. For instance, with a different number of threads at run time, the hints for static data can be inaccurate.

7.3 CONTRIBUTIONS

This thesis makes the following contributions to the state of the art:

- This thesis proposes the software-oriented shared cache management approach for CMPs. The new approach deviates from traditional hardware-based cache management approaches substantially. It opens up a new direction for NUCA cache optimization and provides an alternative solution for system designs.

- Based on the proposed approach, three different management schemes are proposed and studied. The flexibility of the software-oriented shared cache management approach enables many possible design choices. The three proposed management schemes shed light on how this flexible framework can be utilized.

- For single-threaded programs, remote access latency and cache miss rate are regarded as the two most important factors when optimizing program performance. New algorithms are proposed to optimize them simultaneously while most previous studies only consider one of them at a time.

- For latency-oriented multithreaded programs, a novel cache access pattern recognition algorithm is proposed. Instead of capturing detailed data mapping information as proposed in previous works, our algorithm is able to characterize patterns that are independent on program parameters and cache configurations.

- For throughput-oriented multithreaded programs, a hint-guided data replication scheme is proposed and studied. This is the first study on the off-line analysis of data replication characteristics for throughput-oriented multithreaded programs. Our study shows that this type of workloads is very difficult to optimize using off-line analysis hints. The proposed controlled data replication scheme shows its potential by achieving comparable performance to the existing hardware schemes.

7.4 FUTURE WORK

This thesis opens up a new direction for NUCA cache research. The study demonstrates that this is a very promising approach. It has large potential to be explored. Thus there are still many interesting research topics that can be considered as the possible future work:

- The pattern recognition algorithm presented in this thesis is still in its rudimental stage. Only several simple access patterns are studied. To discover more complex data access patterns, more sophisticated algorithms are required. It could bring tremendous returns when more patterns are covered by the proposed cache management scheme.
- The proposed trace analysis scheme for throughput-oriented programs shows the potential of the software-oriented shared cache management approach for this type of workloads. However, the proposed scheme is not significantly better than existing hardware-based cache optimization schemes. Furthermore, using profile-driven optimization method for throughput-oriented programs is barely studied before. Thus it is worth investigating more advanced trace analysis schemes that are able to characterize dynamic activities.
- Given the flexibility of the proposed data placement framework, it will be interesting to study how to optimize other aspects of the system using the same approach. The possible topics include optimizing cache sharing fairness, improving power efficiency of NUCA caches, optimizing bandwidth requirement of on-chip networks, and minimizing the impact of hardware and software faults in caches.

BIBLIOGRAPHY

[1] Susanne Albers and Hisashi Koga. "New On-line Algorithms for the Page Replication Problem," *Journal of Algorithms*, pp.75–96 Volume 27, April 1998.

[2] David H. Albonesi. "Editor in Chief's Message: Truly "Hot" Chips – Do We Still Care?," *IEEE Micro*, 27(2): 4–5, March-April 2007.

[3] Haitham Akkary and Michael A. Driscoll. "A Dynamic Multithreading Processor," *Proceedings of International Symposium on Microarchitecture (MICRO)*, pp. 226–236, November 1998.

[4] Apache HTTP server benchmarking tool `httpd://www.apache.org/`.

[5] AMD Athlon Quad-Core Processors. `http://www.amd.com`.

[6] AMD Phenom Quad-Core Processors. `http://www.amd.com`.

[7] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. "The Landscape of Parallel Computing Research: A View from Berkeley," *Technical Report UCB/EECS-2006-183*, Univ. of California, Berkeley, December 2006.

[8] Todd Austin, Eric Larson, and Dan Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, 35(2):59–67, February 2002.

[9] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. "Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches," *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 250–261, February 2009.

[10] Jean-Loup Baer and Tien-Fu Chen. "Effective Hardware-Based Data Prefetching for High-Performance Processors." *IEEE Computer*, 44(5):609–623, May 1995.

[11] P. Bannon. "Alpha 21364: A Scalable Single-Chip SMP," *Microprocessor Forum*, http://www.digital.com/alphaoem/microprocessorforum.htm October 1998.

[12] Luiz A. Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. “Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 282–293, May 2000.

[13] Bradford M. Beckmann and David A. Wood. “Managing Wire Delay in Large Chip-Multiprocessor Caches” *Proceedings of International Symposium on Microarchitecture (MICRO)*, pp. 319–330, December 2004.

[14] Bradford M. Beckmann, Michael R. Marty and David A. Wood. “ASR: Adaptive Selective Replication for CMP Caches” *Proceedings of International Symposium on Microarchitecture (MICRO)*, pp. 443–454, December 2006.

[15] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. “Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches,” *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 158–170, October 1994.

[16] D. L. Black and D. D. Sleator. “Competitive Algorithms for Replication and Migration Problems,” *Technical Report*, CMU-CS-89-201, Carnegie Mellon University 1989.

[17] W. Bolosky, R. Fitzgerald, and M. Scott. “Simple but Effective Techniques for NUMA Memory Management,” *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, pp. 19–31, 1989.

[18] Shekhar Y. Borkar, Pradeep Dubey, Kevin C. Kahn, David J. Kuck, Hans Mulder, Stephen S. Pawlowski and Justin R. Rattner. “Platform 2015: Intel Processor and Platform Evolution for the Next Decade,” *Tech.@Intel Mag.*, March 2005.

[19] Jeffery A. Brown, Rakesh Kumar and Dean Tullsen. “Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures,” *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

[20] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. “Compiler-Directed Page Coloring for Multiprocessors,” *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 244–255, October 1996.

[21] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales and Michael Shebanow. “Single Instruction Stream Parallelism Is Greater Than Two,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 276–286, May 1991.

[22] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. “Scheduling and Page Migration for Multiprocessor Computer Servers,” *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 12–24, October 1994.

[23] J. Chang and G. S. Sohi. “Cooperative Caching for Chip Multiprocessors,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 264–276, June 2006.

[24] Mainak Chaudhuri “PageNUCA: Selected Policies for Page-grain Locality Management in Large Shared Chip-multiprocessor Caches,” *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 227–238, February 2009.

[25] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. “Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures,” *Proceedings of International Symposium on Microarchitecture (MICRO)*, pp. 55–67, December 2003.

[26] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. “Optimizing Replication, Communication, and Capacity Allocation in CMPs,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 357–368, June 2005.

[27] Sangyeun Cho and Lei Jin. “Managing Distributed, Shared L2 Caches through OS-Level Page Allocation,” *Proceedings of International Symposium on Microarchitecture (MICRO)*, pp. 455–465, December 2006.

[28] A. L. Cox and R. J. Fowler. “The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM,” *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pp. 32–44, December 1989.

[29] David E. Culler, Jaswinder P. Singh and Anoop Gupta. “Parallel Computer Architecture: A Hardware/Softwrare Approach” *Morgan Kaufmann* August 1998, ISBN 1-55860-343-3

[30] G. E. Daddis, Jr. and H. C. Torng. “The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors,” *Proceedings of International Conference on Parallel Processing (ICPP)*, pp. 76–83, August 1991.

[31] William J. Dally and Brian Towles. “Route Packets, Not Wires: On-Chip Interconnection Networks,” *Proceedings of Design Automation Conference (DAC)*, June 2001.

[32] J. A. DeRosa and H. M. Levy. “An Evaluation of Branch Architectures,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 10–16, June 1987.

[33] Haakon Dybdahl and Per Stenstrom. “An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors,” *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 2–12, February 2007.

[34] Rudolf Fleischer and Steven S. Seiden. “New Results for Online Page Replication,” *Proceedings of International Workshop on Approximation Algorithms for Combinational Optimization*, pp. 144–154, September 2000.

[35] Kourosh Gharachorloo, Anoop Gupta, John Hennessy. “Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 22–33, May 1992.

[36] Wlodzimierz Glazek. "Online Algorithms for Page Replication in Rings," *Theoretical Computer Science*, pp. 107–117 Volume 268 Issue 1, October 2001.

[37] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. "A Single-Chip Multiprocessor," *IEEE Computer*, pp. 79–85, September 1997.

[38] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. "The Stanford Hydra CMP," *IEEE Micro*, 20(2): 71–84, March 2000.

[39] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 184–195, June 2009.

[40] J. M. Hart, K. T. Lee, D. Chen, L. Cheng, C. Chou, A. Dixit, D. Greenley, G. Gruber, K. Ho, J. Hsu, N. G. Malur, and J. Wu. "Implementation of a Fourth-Generation 1.8-GHz Dual-Core SPARC V9 Microprocessor," *IEEE Journal of Solid-State Circuits (JSSC)*, 41(1): 210–217, January 2006.

[41] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 3rd Ed., Elsevier, 2003.

[42] M. A. Holliday "Reference History, Page Size, and Migration Daemons in Local/Remote Architectures," *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 104–112, October 1989.

[43] J. Huh, D. Burger, and S. W. Keckler. "Exploring the Design Space of Future CMPs," *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 199–210, September 2001.

[44] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. "A NUCA Substrate for Flexible CMP Cache Sharing," *Proc. Int'l Conf. Supercomputing (ICS)*, pp. 31–40, June 2005.

[45] Intel Quad-Core Processors. `http://www.intel.com`.

[46] Intel. "A New Era of Architectural Innovation Arrives with Intel Dual-Core Processors," *Tech.@Intel Mag.*, May 2005.

[47] Y. Jegou and O. Temam. "Speculative Prefetching," *Proc. Int'l Conf. Supercomputing (ICS)*, pp. 57–66, June 1993.

[48] Lei Jin and Sangyeun Cho "Taming Single-Thread Program Performance on Many Distributed On-Chip L2 Caches," *Proceedings of International Conference on Parallel Processing (ICPP)*, pp. 487–494, Sep. 2008.

[49] Mike Johnson. “Superscalar Microprocessor Design,” *Printice-Hall*, 1991, ISBN 0-13-875634-1

[50] Doug Joseph and Dirk Grunwald. “Prefetching Using Markov Predictors,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 252–263, June 1997.

[51] Norman P. Jouppi. “Improving Directed-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 364–373, June 1990.

[52] R. E. Kessler and M. D. Hill. “Page Placement Algorithms for Large Real-Indexed Caches,” *ACM Transactions on Computer Systems (TOCS)*, 10(4): 338–359, November 1992.

[53] C. Kim, D. Burger, and S. W. Keckler. “An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches,” *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 211–222, October 2002.

[54] Laszlo B. Kish. “End of Moore’s Law: Thermal Death of Integration in Micro and Nano Electronics,” *Physics Letters*, pp. 144–149, December 2002.

[55] Hisashi Koga. “Randomized On-line Algorithms for the Page Replication Problem,” *Proceedings of International Symposium on Algorithms and Computation (ISAC)*, pp. 436–445, 1993.

[56] P. Kongetira, K. Aingaran, and K. Olukotun. “Niagara: A 32-Way Multithreaded Sparc Processor,” *IEEE Micro*, 25(2): 21–29, March-April 2005.

[57] David Koufaty and Deborah T. Marr. “Hyperthreading Technology in the Netburst Microarchitecture,” *IEEE Micro*, 23(2): 56–65, March 2002.

[58] Cameron Laird. “Lightweight Web Servers,” *IBM Development Works*, July 2007.

[59] R. P. LaRowe and C. S. Ellis. “Experimental Comparison of Memory Management Policies for NUMA Multiprocessors,” *ACM Transactions on Computer Systems (TOCS)*, 9(4): 319–363, November 1991.

[60] James Laudon and Daniel Lenoski. “The SGI Origin: A ccNUMA Highly Scalable Server,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 241–251, June 1997.

[61] J. Lee and A. J. Smith. “Branch Prediction Strategies and Branch Target Buffer Design,” *IEEE Computer*, pp. 6–22, January 1984.

[62] David J. Lilja. “Reducing the Branch Penalty in Pipelined Processors,” *IEEE Computer*, 21(7):47–55, July 1988.

[63] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang and P. Sadayappan. "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems," *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, February 2008.

[64] Jaydeep Marathe and Frank Mueller. "Hardware profile-guided automatic page placement for ccNUMA systems," *Proceedings of International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2006.

[65] Michael R. Marty and Mark D. Hill. "Virtual Hierarchies to Support Server Consolidation," *Proceedings of International Symposium on Computer Architecture (ISCA)*, June 2007.

[66] Collin McCurdy and Charles Fischer. "Using Pin as a Memory Reference Generator for Multiprocessor Simulation," *ACM SIGARCH Computer Architecture News*, December 2005.

[67] S. McFarling and J. Hennesey. "Reducing the cost of branches," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 396–403, June 1986.

[68] Steven McGeady. "The i960CA SuperScalar Implementation of the 80960 Architecture," *IEEE*, pp. 232–240 1990.

[69] Gordon E. Moore "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, No. 8, April 1965.

[70] Tomer Y. MOrad, Uri C. Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguade. "Performance, Power Efficiency, and Scalability of Asymmetric Cluster Chip Multiprocessors," *Computer Arhictecture Letters*, vol. 4, July 2005.

[71] S. Naffziger, B. Stackhouse, and T. Grutkowski. "The Implementation of a 2-core Multi-Threaded Itanium-Family Processor," *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 182–183, 592, February 2005.

[72] Basem A. Nayfeh, Kunle Qlukotun and Jaswinder P. Singh. "The Impact of Shared-Cache Clustering in Small-Scale Shared-Memory Multiprocessors," *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 74–84, February 1996.

[73] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson and Kunyung Chang. "The Case for a Single-Chip Multiprocessor," *Proceedings of International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 2–11, September 1996.

[74] OSDL Database Test Suite `http://osdldbt.sourceforge.net/`.

[75] Subbarao Palacharla, Norman P. Jouppi and J. E. Smith. “Complexity-effective Superscalar Processors,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 206–218, May 1997.

[76] Christian Bienia, Kunle Olukotun, and Jaswinder P. Singh. “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 72–81, October 2008.

[77] David A. Patterson and Carlo H. Sequin. “RISC I: A Reduced Instruction Set VLSI Computer,” *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 443–457, May 1981.

[78] Moinuddin K. Qureshi and Yale N. Patt. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Run-Time Mechanism to Partition Shared Caches,” *Proceedings of International Symposium on Microarchitecture (MICRO)*, pp. 423–432, December 2006.

[79] Moinuddin K. Qureshi. “Adaptive Spill-Receive for Robust High-Performance Caching in CMPs,” *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 45–54, February 2009.

[80] R. M. Ramanathan. “Intel Multi-Core Processors: Making the Move to Quad-Core and Beyond,” *Intel White Paper*, 2006.

[81] Justin R. Rattner “Tera-scale Computing: A Parallel Path to the Future,” *Intel Software Network*, May 2007

[82] T. Sherwood, B. Calder, and J. Emer. “Reducing Cache Misses Using Hardware and Software Page Placement,” *Proceedings of International Conference on Supercomputing (ICS)*, pp. 155–164, June 1999.

[83] Timothy Sherwood, Suleyman Sair and Brad Calder. “Predictor-Directed Stream Buffers,” *Proceedings of International Symposium on Microarchitecture (MICRO)*, pp. 42–53, December 2000.

[84] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. “POWER5 system microarchitecture,” *IBM Journal of Research and Development*, 49(4/5):505–521, July/September 2005.

[85] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. “Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors,” *Proceedings of International Symposium Computer Architecture (ISCA)*, pp. 346–356, June 2005.

[86] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh and Anoop Gupta. “The SPLASH-2 Programs: Characterization and Methodological Considera-

tions," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 24–36, June 1995.

[87] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware," *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 255–266, November 1994.

[88] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Michael Abrash, Jeremy Suqerman, and Pat Harahan. "Larrabee: A Many-core x86 Architecture for Visual Computing," *IEEE Micro*, 29(1): 10–21, January 2009.

[89] Virtutech AB. Simics Full System Simulator. `http://www.simics.com/`.

[90] James E. Smith. "A Study of Branch Prediction Strategies," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 135–148, May 1981.

[91] James E. Smith and Gurindar S. Sohi. "The Microarchitecture of Superscalar Processors," *Proceedings of The IEEE* 83(12): 1609–1624, December 1995.

[92] Luke Reeves. "Cheetah," *http://www.neuro-tech.net/cheetah*

[93] John Shen and Mikko Lipasti "Modern Processor Design: Fundamentals of Superscalar Processors" *McGraw-Hill* 2005, ISBN 0-07-057064-7

[94] Vijayaraghavan Soundararajan, Mark Heinrich, Ben Verghese, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. "Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 342–355, June 1998.

[95] Standard Performance Evaluation Corporation. SPEC 2000 and SPECjbb 2005. `http://www.specbench.org`.

[96] Sun Microsystems. Solaris OS. `http://www.sun.com/software/solaris`.

[97] Sun Microsystems. Sun UltraSPARC Processor. `http://www.sun.com/processors`.

[98] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. "Accelerating Critical Section Execution with Asymmetric Multi-core Architectures," *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 253–264, October 2009.

[99] M. Aater Suleman, Yale N. Patt, E. Sprangle, A. Rohillah, A. Ghuloum, and D. Carmean. "ACMP: Balancing Hardware Efficiency and Programmer Efficiency," *HPS Technical Report*, TR-HPS-2007-001 February 2007.

[100] Steven Swanson, Luke K. McDowell, Michael M. Swift, Susan J. Eggers and Henry M. Levy. "An Evaluation of Speculative Instruction Execution on Simultaneous Multithreaded Processors," *ACM Transactions on Computer Systems (TOCS)*, 21(3): 314–340, August 2003.

[101] T. Takayanagi *et al.* T. Takayanagi, J. L. Shin, B. Petrick, J. Y. Su, H. Levy, H. Pham, J. Son, N. Moon, D. Bistry, U. Nair, M. Singh, V. Mathur, and A. S. Leon. "A Dual-Core 64-bit UltraSPARC Microprocessor for Dense Server Applications," *IEEE Journal of Solid-State Circuits (JSSC)*, 40(1):7–18, January 2005.

[102] David Tam, Reza Azimi, Livio Soares and Michael Stumm. "Managing Shared L2 Caches on Multicore Systems in Software," *In Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, October 2007.

[103] TILE64 family of multicore processors. `http://www.tilera.com`.

[104] Mustafa M. Tikir and Jeffery K. Hollingsworth. "NUMA-Aware Java Heaps for Server Applications," *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 108.2, 2005.

[105] Dean M. Tullsen, Susan J. Eggers and Henry M. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 392–403, May 1995.

[106] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," *Proceedings of International Conference on Architectural Support for Proggramming Languages and Operating Systems (ASPLOS)*, pp. 279–289, October 1996.

[107] David W. Wall. "Limits of Instruction-Level Parallelism," *Proceedings of International Conference on Architectural Support for Proggramming Languages and Operating Systems (ASPLOS)*, pp. 176–188, April 1991.

[108] K. M. Wilson and B. B. Aglietti. "Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C," *Proceedings of ACM/IEEE Conference on Supercomputing (SC)*, pp. 258–265, November 2001.

[109] Kenneth C. Yeager. "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, 16(2): 28–40, April 1996.

[110] Tse-Yu Yeh and Yale N. Patt. "Two-Level Adaptive Training Branch Prediction," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 51–61, June 1991.

[111] Michael Zhang and Krste Asanović. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 336–345, June 2005.