國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

PARSEC 與 SPLASH-2 之工作特性分析

Workload Characterization of PARSEC and SPLASH-2

林怡賢

I-Hsien Lin

指導教授：楊佳玲 博士

Advisor: Chia-Lin Yang, Ph.D.

中華民國 99 年 7 月

July, 2010

摘要

在這篇論文中，我們首先試著去分析PARSEC與SPLASH-2這兩個常用的評測基準程式(benchmark suite)之工作特性。每個工作量(workload)都會在各種不同的模擬機器配置下進行實驗來研究他們多樣化的特性。此外，我們也試著啟動一個軟體預取機制(software prefetching scheme)來評估對這些不同工作量所造成的影響。根據這些實驗，我們發現到當執行這些工作量時，在互連網路上常常會有大量對共享資料做讀取動作的傳輸。

針對平行程式中大量的共享資料讀取失誤，我們提出一個稱為Snooping Fetch (SF)的方法。這個方法藉由系統中固有的快取一致性協定(cache coherence protocol)以及監聽協定(snooping protocol)所廣播的資訊來試著減少這些共享資料的讀取失誤。我們可以利用這個廣播的特性，在一筆資料真正要被使用之前先把資料讀進快取記憶體中以隱藏讀取時的延遲。在我們的實驗中，這個方法針對減少L1快取記憶體失誤量的部分可以得到平均8%的可能性，而在L1快取失誤率的減少量約有平均0.5%的可能性，以及平均0.4%的效能加速可能性。

關鍵字：PARSEC、SPLASH-2、評測基準程式、工作特性分析、預取、隱藏資料存取延遲

Abstract

In this thesis, we first characterize two popular benchmark suites, PARSEC and SPLASH-2. Different configurations of the simulated machine are set for each workload in the suites to study their diverse properties. Besides, we also try to evaluate the influence of different workloads when enabling a software prefetching scheme. Based on the experiments, we find that there is plenty of shared read traffic on the interconnection in these workloads.

To reduce these shared read misses, a policy, called *Snooping Fetch (SF)*, is proposed for the shared data in parallelized applications, which takes advantage of the inherent cache coherence protocol and the broadcasting information from the snooping protocol. The SF policy utilizes the broadcasting property to fetch shared data before the data is actually needed. In our experiments, the SF policy provides a reduction potential of 8% in average in the amounts of L1 cache misses, a reduction potential of 0.5% in average in the L1 cache miss rate, and a potential of 0.4% speedup in average.

Keywords — PARSEC, SPLASH-2, benchmark suite, workload characterization, prefetch, data access latency hiding

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When designing an efficient system or doing computer architecture research, benchmark as a standard for quantitative evaluation is vital. Many benchmark suites are released to provide different requirements of applications. For the resent years, more and more processors are put into one machine for the growth of computation power and the reduction of power consumption. Applications that require additional processing power will need to be parallelized. The behaviour of parallel programs is greatly different from the earlier serial applications. Therefore, it is significant to have several parallel applications as benchmarks when studying and designing such kind of systems. PARSEC[2][3] and SPLASH-2[18] benchmark suites release lots of parallel applications for the multiprocessor system. They are widely used for the verification of new ideas and the tradeoffs in the system. SPLASH-2 suite focuses on High-Performance Computing for shared-address-space multiprocessors; PARSEC suite provides a wide range of applications for the studies of Chip-Multiprocessors (CMPs). To better understand these two suites, we try to characterize the workloads belong to them. In this work, we concentrate on the L1 cache configurations for all the workloads. Besides, we also want to observe the influence of prefetching. A software prefetching policy is applied to compare the difference of cache miss rate. Moreover, since these workloads are all multi-threaded programs, the interaction between different threads is a major issue. We evaluate the number of sharers and the amount of messages which are placed on the interconnection to get a detailed information of the workloads in the PARSEC and SPLASH-2 suites.

Based on the observation about the characterization of these workloads, we find that there is around 22% to 32% shared read traffic over the total amount of the interconnection traffic. It also implies that there are lots of shared read misses. To reduce these shared read misses, we then propose *Snooping Fetch (SF)*, a policy attempts to utilize the coherence state and the broadcasting information on the interconnection to fetch data in advance without demanding extra bandwidth. In a snooping protocol, all the requests and data are broadcast on the interconnection. When the processor snoops the interconnection, the shared data is likely to be seen before the data is actually needed. If we can fetch the data in advance, the following request for this data will become a cache hit instead of a cache miss. Therefore, the number of misses can be decreased. Besides, the activity of fetching the shared data on the interconnection does not incur any extra transmissions, so there is no bandwidth overhead. The inherent coherence protocol is employed to identify the shared data. Moreover, we add a buffer for each processor to gain more potential in the SF policy. These buffers store the data fetched by our SF policy. To evaluate the potential of the SF policy, the buffers are set as unlimited size to place the data on the interconnection. We first use a experiment of traffic breakdown to verify that the SF policy surely reduces the shared read misses. And we adjust different parameters to evaluate the improvement potential of the SF policy. Our SF policy can achieve a reduction potential about 8% in the amounts of cache misses over baseline and a potential of 0.4% speedup over baseline.

This paper is organized as follows. The next Chapter 2 provides a rough introduction to the workloads in PARSEC and SPLASH-2 benchmark suites. Chapter 3 shows the configuration parameters used in the simulation. Chapter 4 presents the experiment results of each workload with different system settings and highlights the diverse characterization of these workloads. Chapter 5 introduces the SF policy which relies on the snooping protocol and the inherent coherence protocol to reduce shared read misses. Chapter 6 gives a comparison between the original policy and the SF policy. It also presents the improvement potential of our approach. An overview of related works in the workload characterization and previous studies of hiding data access latency are reported in Chapter 7. Finally, Chapter 8 gives conclusions.

Chapter 2

Workloads

2.1 PARSEC Suite

PARSEC is a benchmark suite for studies of Chip-Multiprocessors (CMPs). It tries to assemble a program selection that is large and diverse enough to be sufficiently representative for scientific studies. The suite includes not only a number of important applications from the RMS suite but also several leading-edge applications from Princeton University, Stanford University and the open-source domain.

blackscholes is an Intel RMS workload. It calculates the prices for a portfolio of options. The program is chosen to represent the field of analytic partial differential equation (PDE)[4] solvers especially in computational finance and is limited by the amount of floating-point calculations a processor can perform.

bodytrack is an Intel RMS workload. It tracks the pose of a marker-less human body with multiple cameras. An annealed particle filter is employed to track the pose using edges and the foreground silhouette as image features. The program has four parallel kernels: edge detection, edge smoothing, particle weights calculation and particle resampling.

canneal is developed by Princeton University. It employs a simulated annealing (SA) algorithm to minimize the routing cost of a chip design. This optimization method tries to pseudo-randomly swap netlist elements. If the swap would decrease the routing cost, it is automatically accepted. With a certain probability that is decreasing over time, a swap that would increase the routing cost is also accepted so that the design can escape from local minima. The program has a demanding memory access behaviour and also tracks the swaps globally which increases communication between threads.

ferret is developed by Princeton University. It is a search engine which finds a set of images similar to a query image by analyzing their contents. The program represents emerging next-generation desktop and Internet search engines for non-text document data types. It is parallelized using the pipeline model with six stages. The first and the last stage are for input and output. The middle four stages are for query image segmentation, feature extraction, indexing of candidate sets and ranking. Each stage has its own thread pool and the basic work unit is a query image.

fluidanimate is an Intel RMS workload. It simulates the underlying physics of fluid motion for real-time animation purposes and its output can be visualized by detecting and rendering the surface of the fluid. Every time step, the program executes five kernels: rebuild spatial index, compute densities, compute forces, handle collisions with scene geometry and update positions of particles.

steamcluster is developed by Princeton University. It finds a predetermined number of medians so that each point is assigned to its nearest center for a stream of input points. The program is a common operation where large amounts of continuously produced data has to be organized under real-time conditions and spends most of its time evaluating the gain of opening a new center. This operation uses a parallelization scheme which employs static partitioning of data points. It is memory bound for low-dimensional data and becomes increasingly computationally intensive as the dimensionality increases.

swaptions is an Intel RMS workload. It prices a portfolio of swaptions. The program is included because of the significance of partial differential equation (PDE)[4] and the wide use of Monte Carlo simulation. The portfolio array is partitioned into a number of blocks equal to the number of threads and the block is assigned to every thread. Each thread iterates through all swaptions in the assigned work unit and computes the price.

vips was originally developed through several projects funded by European Union (EU) grants. The benchmark version is derived from a print on demand service that is offered at the National Gallery of London. It is an image processing system and includes fundamental image operations. The program fuses all image operations to construct an image transformation pipeline that can operate on subsets of an image and it can also automatically replicate the image transformation pipeline to process multiple image regions concurrently.

x264 is a H.264/AVC (Advanced Video Coding) video encoder. Motion compensation scheme is used to detect and eliminate data redundancy. It is employed to exploit temporal redundancy between successive frames. Motion compensation is usually the most expensive operation that has to be executed to encode a frame.

2.2 SPLASH-2 Suite

The SPLASH-2 suite is released to facilitate the study of shared-address-space multiprocessors. It is expanded and modified to include several new programs as well as improved versions of the original Stanford ParalleL Applications for SHared memory (SPLASH)[15] programs. The resulting SPLASH-2 suite contains programs that represent a wider range of computations in the scientific, engineering and graphics domains.

barnes simulates the interaction of a system of bodies in three dimensions over a number of time-steps, using the Barnes-Hut N-body method. In this program, most of the time is spent in partial traversals of the octree to compute the forces on individual bodies. The communication patterns are dependent on the particle distribution and are quite unstructured.

fmm simulates the interaction of a system of bodies in two dimensions over a number of time-steps, using a N-body method called the adaptive Fast Multipole Method. The major data structures are body and tree cells, with multiple particles per leaf cell. In this program, the tree is not traversed once per body, but only in a single upward and downward pass that computes interactions among cells and propagates their effects down to the bodies. The communication patterns are also quite unstructured.

lu factors a dense matrix into the product of a lower triangular and an upper triangular matrix. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. The block size should be large enough to keep the cache miss rate low, and small enough to maintain good load balance. Elements within a block are allocated contiguously to improve spatial locality and blocks are allocated locally to processors that own them.

ocean studies large-scale ocean movements based on eddy and boundary currents. It partitions the grids into square-like subgrids rather than groups of columns to improve the communication to computation ratio. Grid computations in the same horizontal section are independent of one another and those in the same vertical section follow a thread of dependence.

radix is a parallel sorting algorithm. The program is iterative, performing one iteration for the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication.

water evaluates forces and potentials that occur over time in a system of water molecules. It imposes a uniform 3-D grid of cells on the problem domain. Processors which own a cell need only look at neighboring cells to find molecules that might be within the cutoff radius of molecules in the box it owns. The movement of molecules into and out of cells causes cell lists to be updated and therefore resulting in communication.

Chapter 3

Simulation Setup

We have evaluated PARSEC and SPLASH-2 benchmark suites with Simics[12], a full-system execution-driven simulator, for 8 processors with SPARC systems running Solaris 10. The Simics simulator extends with the GEMS[13] toolset that provides a detailed memory hierarchy model and a detailed processor model. We attach a cache hierarchy of a private L1 cache for each processor and a large shared L2 cache. The cache model applies a MOSI coherence protocol to maintain the consistency of the data. Since bus interconnection is still widely used in recent multiprocessor systems. We also implement a bus interconnection in GEMS for our workload characterization. The traffic on the bus between L1 and L2 caches what we called *L1-L2 bus* is observed in the following experiments to extract the properties of different workloads. The configuration parameters are summarized in Table 3.1. These settings are applied in all of the experiments only except that we mention in the article specifically.

We use the simlarge input set for PARSEC, and a recommended input

Table 3.1: Simulation Setup

Processor Number	8
Block Size	64 bytes
L1 Cache	64 KB, 4-way, 2 cycle access time
L2 Cache	4 MB, 4-way, 25 cycle access time
Coherence Protocol	MOSI protocol
Network Type	Bus

set size for SPLASH-2. A detailed configuration is showed in Table 3.2. For both benchmark suites, we warm all the caches during the initial 100 million instructions, and then perform a detailed simulation for the next 200 million instructions.

In the following sections, there are two abbreviations used in the illustrating figures.

(1) “p” and “np”: “p” represents that the prefetch instructions decoding in GEMS is enabled; “np” represents that the prefetch instructions decoding in GEMS is disabled

(2) “BASE” and “SF”: “BASE” represents that the result is under the original policy; “SF” represents that the result is under our SF policy

Table 3.2: Input Set of PARSEC and SPLASH-2 suites

Benchmark	**Input Set**
blackscholes	65,536 options
bodytrack	4 cameras, 4 frames, 4,000 particles, 5 annealing layers
canneal	15,000 swaps per temperature step, 400,000 netlist elements
ferret	256 image queries, database with 34,973 images, find top 10 images
fluidanimate	300,000 particles, 5 frames
streamcluster	16,384 input points, 128 point dimensions
swaptions	64 swaptions, 20,000 simulations
vips	2,662 5,500 pixels
x264	640 360 pixels, 128 frames
barnes	16,384 particles
fmm	16,384 particles
lu	512 512 matrix, 16 16 blocks
ocean	258 258 grid
radix	1M integers, radix 1024
water	512 molecules

Chapter 4

Workload Characterization

We try to explore the characterization for both PARSEC and SPLASH-2 suites in this chapter. We will examine different factors for each workload in the following experiments, including L1 cache miss rate, sharing degree of data and traffic breakdown of the L1-L2 bus. In these experiments, different parameters are also applied, like L1 cache size, cache line size and number of cores.

In the first two sections, the L1 cache miss rate is observed to explore the property of each workload. Moreover, we also analyze the effect of the software prefetching policy done by the compiler. Software prefetching executes prefetch instructions inserted by compiler to move data close to the processor in advance. In our experiments, the prefetch instructions are added by gcc compiler. The software prefetching policy in gcc is mainly for loop prefetching. We switch on or off the software prefetching scheme by enabling or disabling the prefetch instruction decoding in the simulator.

Besides, the sharing behaviours in the multi-threaded programs and the amount of traffic on the L1-L2 bus are evaluated. In a multiprocessor system, the sharing behaviours cause a number of coherence activities. We calculate the ratio of shared lines in L2 cache for each workload and distinguish the shared lines based on how many different processors have this data in its L1 cache. The number of processors is called sharing degree. A higher sharing degree implies that the program is likely to incur more transmissions to maintain the cache consistency. Since the memory accesses and extra coherence requests may cause a bus contention and lead to a worse performance. We observe the bus traffic between L1 and L2 caches as well as categorize the traffic according to the type of the cache requests placed on the L1-L2 bus.

4.1 L1 Cache Size

We illustrate the L1 cache miss rate with different cache size in Figure 4.1 and Figure 4.2. It's obvious that we have a lower cache miss rate with the increasing of L1 cache size. It is easy to understand that if there is a bigger cache, we can store more data in the cache and the probability of finding the requested data in the cache becomes higher. From the comparison of these two figures, we can also find that PARSEC suite has a higher L1 cache demand in our setting than SPLASH-2 suite.

We then compare the difference with switching the prefetching scheme on or off. The figures show that enabling the prefetching scheme brings advantage to cache miss rate in most cases. Although several benchmarks in SPLASH-2, including `barnes`, `fmm`, `lu` and `water`, don't get much benefits from the prefetching scheme because of their lower cache miss rate. In

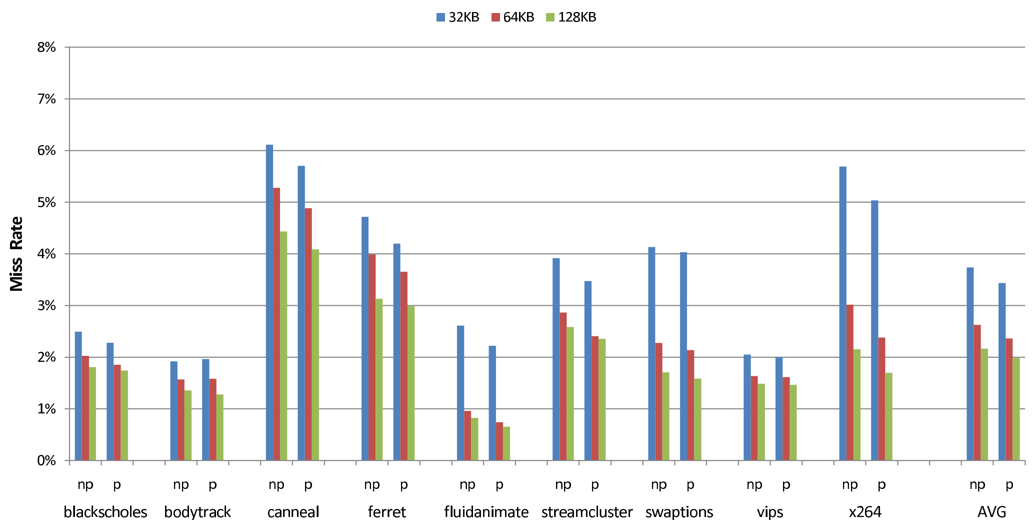

Figure 4.1: L1 Cache Miss Rate of PARSEC Benchmark Suite with the Switching of Prefetching Scheme (np for prefetching disabled; p for prefetching enabled) in Different Cache Size

`bodytrack` and `water`, there is even a higher cache miss rate in smaller cache size when turning on the prefetching scheme. The data is aggressively competing the limited resource as the cache size is small, so prefetching data may easily cause a cache pollution problem and damage the performance.

4.2 Cache Line Size

Figure 4.3 and Figure 4.4 show the L1 cache miss rate with different line size in this section. As we can see from these two figures, the cache miss rate benefits from a larger line size. The augmentation of the line size can exploit the spatial locality in the applications. Applying a larger cache line

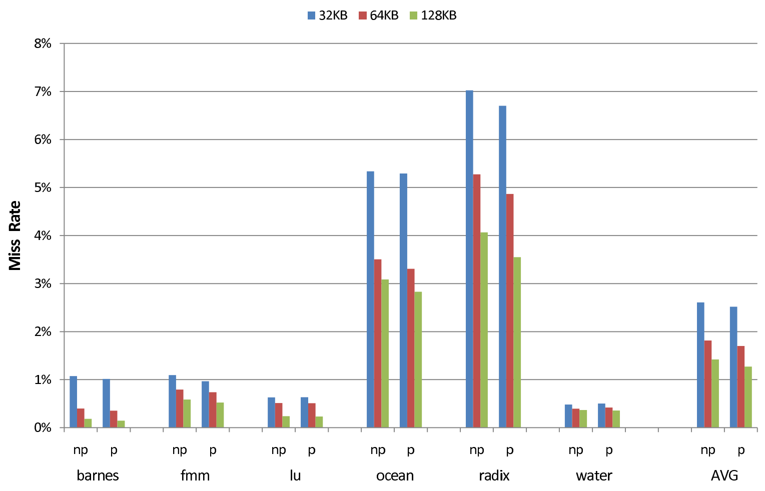

Figure 4.2: L1 Cache Miss Rate of SPLASH-2 Benchmark Suite with the Switching of Prefetching Scheme (np for prefetching disabled; p for prefetching enabled) in Different Cache Size

seems like we implement a hardware next-line prefetching policy. Therefore, we have less cache miss rate as the line size increases. `fluidanimate` and `streamcluster` gain much more improvements than other workloads because these two workloads have a streaming behaviour and hence provide more spatial locality.

We can find that switching on the prefetching scheme still has advantage in each line size setting. With the increasing of the line size, improvements from the prefetching scheme diminish. Since a larger line size can exploit more spatial locality, the prefetching scheme relatively gets less opportunities to issue prefetch requests. Most of the SPLASH-2 workloads still have few improvements from the prefetching scheme due to their lower cache miss rate. Only the workloads with higher cache miss rate, like `ocean` and `radix`, have a clearer effect in the experiment.

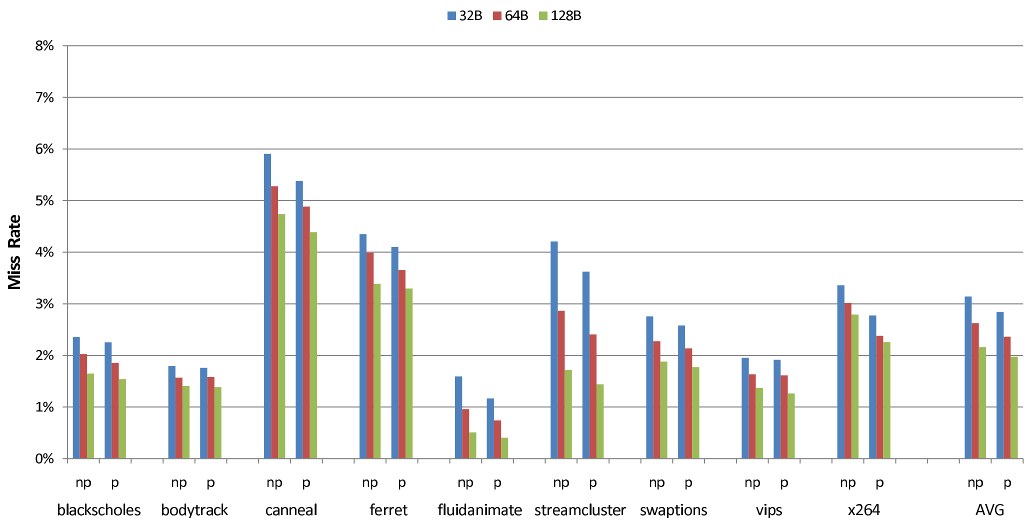

Figure 4.3: L1 Cache Miss Rate of PARSEC Benchmark Suite with the Switching of Prefetching Scheme (np for prefetching disabled; p for prefetching enabled) in Different Line Size

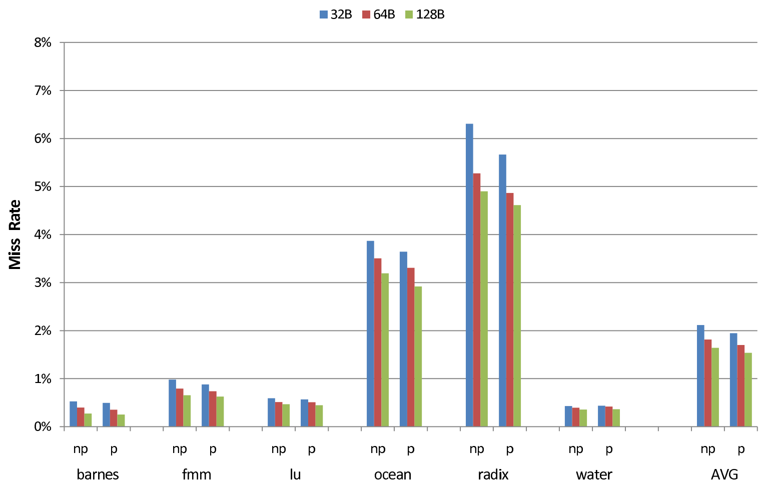

Figure 4.4: L1 Cache Miss Rate of SPLASH-2 Benchmark Suite with the Switching of Prefetching Scheme (np for prefetching disabled; p for prefetching enabled) in Different Line Size

4.3 Sharing Degree

The sharing degree can be considered more particularly in two aspects. One is to examine how many processors have ever accessed this shared data along the total execution, and the other is to check how many processors have this shared data in their local caches sometime during the execution with a constrained cache capacity. The former aspect shows the characterization of a program, and it has nothing to do with the hardware structure. The latter one gives that there are how many caches to cohere at the same time in a program under a specific cache organization.

We first evaluate the total sharers of a program along the whole execution time. And we also adjust the cache line size to observe the effects on the sharing degree. In Figure 4.5 and Figure 4.6, the sharing degree of PARSEC and SPLASH-2 suites is demonstrated. As our expectation, we can see that the sharing degree grows as line size increases in these two figures. Since a cache block holds a larger data, it provides a higher possibility to be accessed by more processors.

However, we get a different result of sharing degree if we calculate the number of processors which have the shared data in their local cache at the same time with a constrained L2 cache capacity. The sharing degree with a constrained cache capacity is presented in Figure 4.7 and Figure 4.8. We can find that the sharing degree of these workloads doesn't always grow while we have a larger line size. This is due to the eviction of the shared data in a limited cache size.

We then focus on the different properties of these workloads by observing

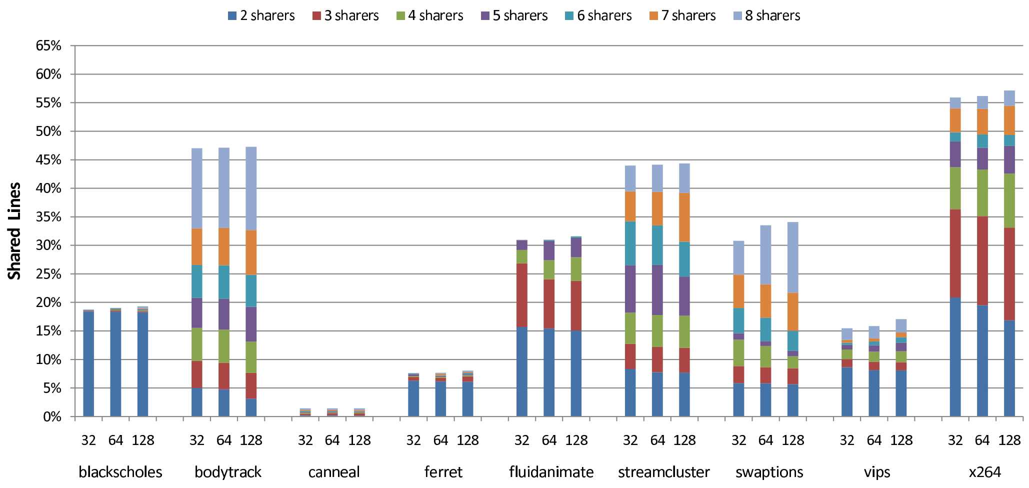

Figure 4.5: Sharing Degree of PARSEC Benchmark Suite with Different Line Size

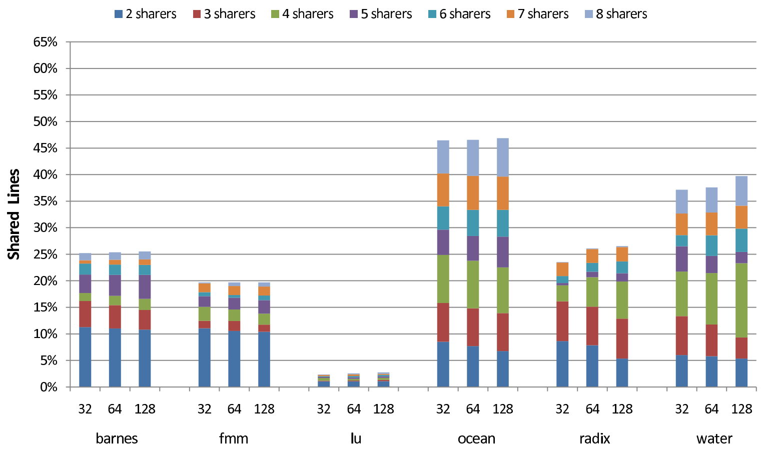

Figure 4.6: Sharing Degree of SPLASH-2 Benchmark Suite with Different Line Size

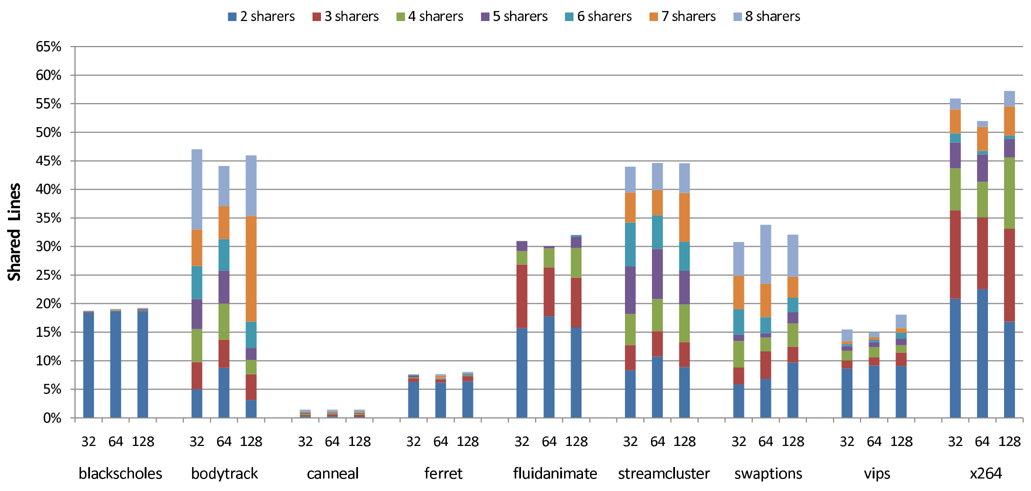

Figure 4.7: Sharing Degree of PARSEC Benchmark Suite in a Constrained Cache Capacity with Different Line Size

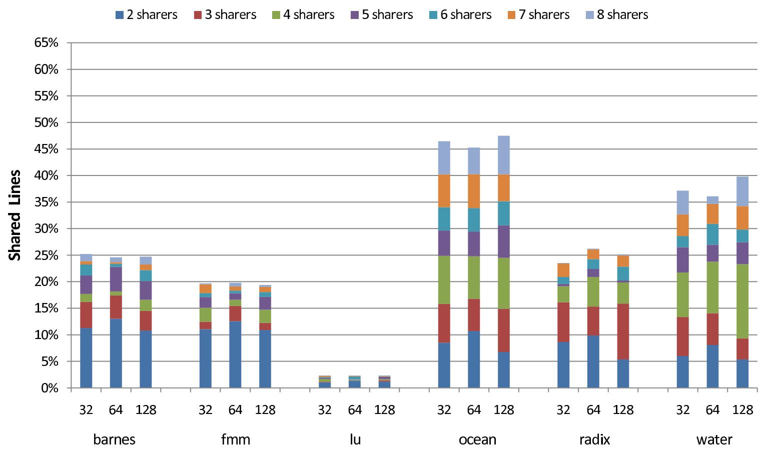

Figure 4.8: Sharing Degree of SPLASH-2 Benchmark Suite in a Constrained Cache Capacity with Different Line Size

their sharing behaviours. `canneal` and `lu` show only trivial amount of sharing. `canneal` has a hunger for cache capacity because of its large working set. So the sharing is limited due to its frequent data exchange in the cache. Since the data stays in the cache for a very short time, there are only a few chances to have sharing behaviours between processors. `lu` divides a dense matrix into an array of small blocks and exploits temporal locality. Blocks are updated by the processors that own them to reduce the communication. Therefore, each processor executes its working set independently and incurs little sharing. Except those two workloads we mentioned above, other workloads present lots of sharing behaviours. The sharing in most benchmarks is around 20% to 50%. `bodytrack`, `streamcluster`, `x264`, and `ocean` present a large amount of sharing. `x264` shows the most significant amounts of sharing due to its motion compensation scheme. Reference frames are shared for the encoding of other frames. However, most of the shared data are only shared by no more than 4 processors. `blackscholes` also presents a number of sharing, but almost all the shared data are only accessed by two processors. The data is shared between the parent thread and its child thread. `ferret` breaks an image into several non-overlapping segments and performs a image similarity search in its database. Besides, the database is scanned by all threads to find entries similar to the query image and the size of the database is practically unbounded. So the cache line is unlikely to be accessed more than once and it is easy to be replaced. Therefore, there is a small amount of sharing in this workload. Moreover, most of the sharing in this workload is also between two processors.

4.4 Traffic Breakdown

In this section, we try to explore the traffic of the bus interconnection between the private L1 caches and the shared L2 cache. The amount of traffic is evaluated and decomposed into four categories by the type of cache misses to observe the ratio of different kind of misses.

(1) Private Write: a write request for a cache line which has no sharer

(2) Shared Write: a write request for a cache line which has one or more sharers

(3) Private Read: a read request for a cache line which has no sharer

(4) Shared Read: a read request for a cache line which has one or more sharers

The sharer we mention here applies the sharing definition with a constrained cache capacity. The reason is that it provides a actual coherence behaviour during the execution. Figure 4.9 and Figure 4.10 show the traffic breakdown of the L1-L2 bus with different number of cores. In `canneal` and `lu`, the ratio of sharing traffic is extremely low. This is because these two workloads have a small amount of sharing as showed in Figure 4.7 and Figure 4.8. The other workloads provide plenty of sharing traffic, especially in `streamcluster`. From these two figures, we can find that there is about 22% to 32% shared read traffic in average. It also means that there is around 22% to 32% shared read misses are issued to the L1-L2 bus. We try to figure out

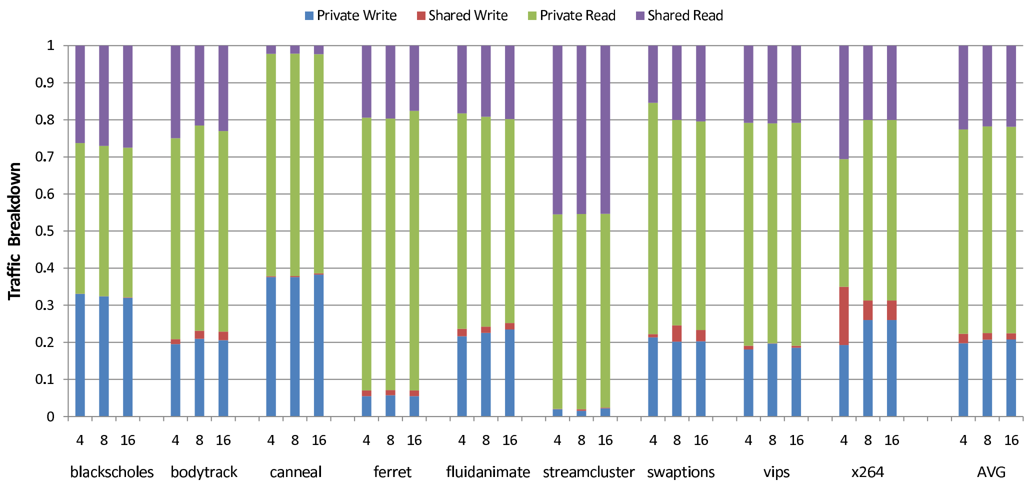

Figure 4.9: Traffic Breakdown of PARSEC Benchmark Suite with Different Number of Cores

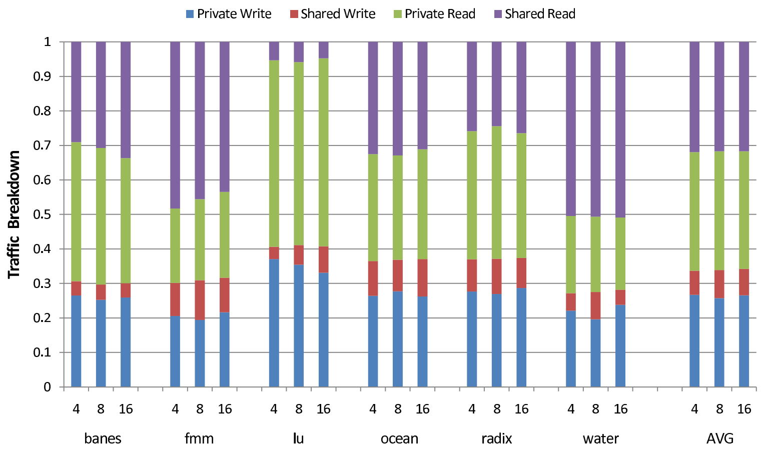

Figure 4.10: Traffic Breakdown of SPLASH-2 Benchmark Suite with Different Number of Cores

a way to reduce the large amounts of shared read misses and thus improve the workload performance.

Chapter 5

Snooping Fetch

In Section 4.3 and Section 4.4, we observe that there are plenty of shared read misses in both PARSEC and SPLASH-2 suites. However, all the requests will be placed on the bus and broadcast to all processors in a snooping protocol. Each processor snoops the request on the bus to see if the requested data is in its own cache. If so, there might be some cache state transitions to maintain cache coherence according to its inherent coherence protocol. Otherwise, the processor ignores this request. There might be some potential to better utilize these information. The shared data is likely to be seen on the bus before the processor actually requests this data. There is no extra bandwidth overhead when fetching data on the bus in addition. To reduce the amounts of shared read misses, we propose a policy, called *Snooping Fetch (SF)*, which utilizes the broadcasting information in the snooping protocol.

We try to identify the shared data in our SF policy. The inherent coherence state is used for indication. In MOSI coherence protocol used in our experiments, "I" state means this cache line is invalid. To enter this state, this cache line should be either read or write by its local processor and then there is one other processor wants to modify this data. Write miss will be sent by that processor and the shared lines in others' caches will be invalidated. So the "I" state can imply that this data is once shared between at least two processors. There is another benefit when using "I" state indication. Fetching data into an invalidated cache line won't need any eviction and hence no useful data will be dropped. Therefore, no cache pollution issue will be incurred. We use a simple example to explain our idea.

For instance, One-Producer-Multiple-Consumer Sharing would benefit from the SF policy as Figure 5.1 and Figure 5.2 present. In the first place, all these 4 processors have one shared data. When the producer tries to modify the data(Figure 5.1① and Figure 5.2①), the shared data is invalidated in all the consumers for both original and SF policies(Figure 5.1② and Figure 5.2②). While the consumers try to read the modified data in sequence, a read miss will be issued for each consumer(Figure 5.1③⑤⑦) and therefore lots of read misses will be incurred in the original policy. Corresponding to each read miss, one consumer fetches the data into its local cache at a time(Figure 5.1④⑥⑧). In SF policy, the modified data will be placed on the bus when the first consumer requests this data(Figure 5.2③). After snooping the bus, all the other consumers will find that their local L1 cache has the same tag as the data on the bus and this cache line is in "I" state. So they will fetch the modified data into its own cache from the bus in advance(Figure 5.2④). Like Figure 5.2 shows, the number of read misses will be reduced because the following read requests from the consumers will hit in its local cache(Figure 5.2⑤⑥).

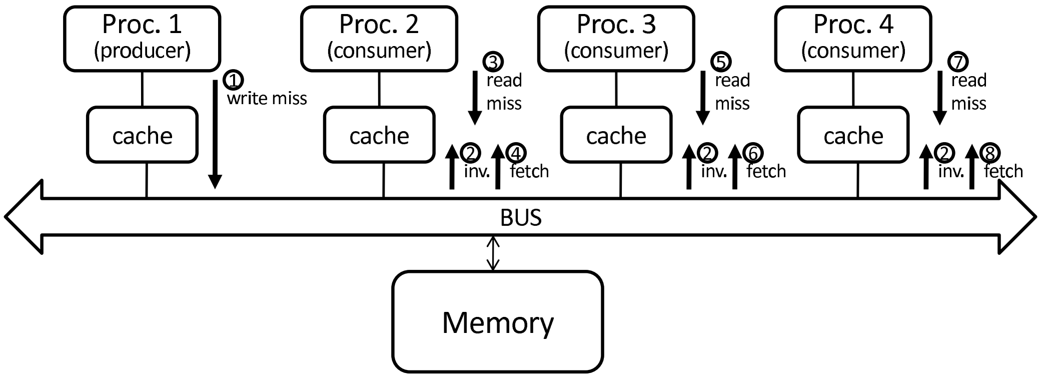

Figure 5.1: An Example of Original Policy

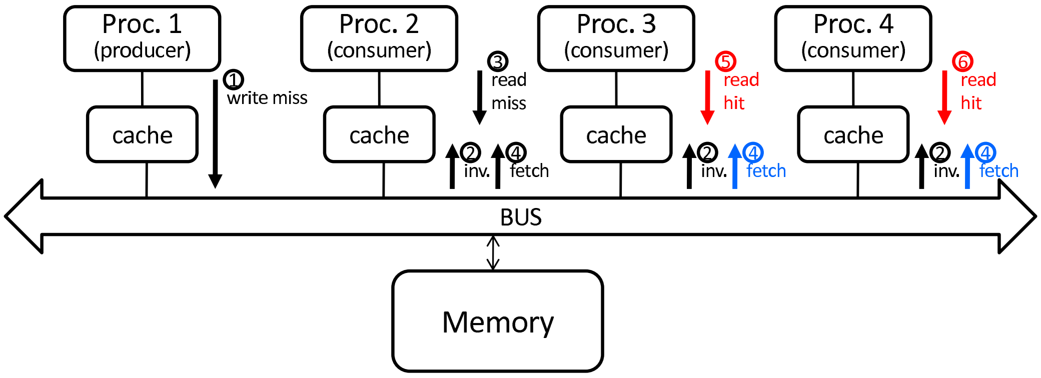

Figure 5.2: An Example of SF Policy

Although fetching the invalidated data can hide the data access latency without incurring any overhead. The potential of SF policy is extremely limited because of the small portion of the invalidated cache lines in L1 caches. To settle this problem, we add a buffer, called *Snooping Fetch Buffer (SFB)*, besides the L1 cache for each processor. The SFB is acted as a cache, but it only stores the data fetched by the SF policy. Since a larger space is available to place these fetched data, we now focus on not just the invalidated data but all kinds of data on the bus. This is because that the invalidated data only occupies a small fraction of the shared data. The invalidated data is still fetched into the L1 cache as the primary idea, and all the other data on the bus is fetched into the SFB. When these data are requested by the processor, there is a L1 cache hit or a SFB hit instead of a L1 cache miss. Furthermore, we should not fetch all the data on the bus, the type of request should also be taken into consideration. If a write request is placed on the bus, it means that this data is gong to be modified, so fetching this data is meaningless. Therefore, SF policy should function only when the request on the bus is a read request.

To reduce the shared read misses, our SF policy tries to fetch the shared data in advance as soon as the data is placed on the bus. In SF policy, adding a SFB for each L1 cache incurs a storage overhead. But the bandwidth issue is eliminated since we don't issue any additional requests and there is no extra data transmission.

Chapter 6

Evaluation of SF Policy

We use SFBs with unlimited size to examine the potential of the proposed SF policy. Besides, based on the previous workload characterization in Chapter 4, we classify the workloads in PARSEC and SPLASH-2 suites into different categories.

(1) *Small amount of sharing (SS)*: Those workloads present a small amount of sharing, including `canneal`, `lu`

(2) *Large amount of sharing (LS)*: Those workloads present a large amount of sharing, including `bodytrack`, `streamcluster`, `x264`, `ocean`

(3) *Sharing between two sharers (TS)*: The shared data in those workloads is almost only accessed by two processors, including `blackscholes`, `ferret`

(4) *Low L1 cache miss rate (LM)*: Those workloads have a quite low L1 cache miss rate, including `barnes`, `fmm`, `water`

(5) *Other workloads (Other)*: Those workloads do not have the above-mentioned properties, including `fluidanimate`, `swaptions`, `vips`, `radix`

There is a clearer demonstration in the following experiments by applying these classifications.

6.1 Traffic Breakdown Comparison

To check the validity of our approach, we compare the traffic breakdown between the original policy and the proposed SF policy. The experiment is like what we present in Section 4.4 except that we take away the portion of write traffic. The reason is that all write instructions will issue a write miss to invalidate all the copies except the requested data is already in its local cache and also the cache line is in a modified state. Since our SF policy only functions when meeting a read request and all the fetched data will be in a shared state, we won't get any advantage from those write misses. Therefore, the write traffic won't be affected by SF policy and we only focus on the effects in the read traffic.

Figure 6.1 and Figure 6.2 show the traffic breakdown under two policies with different number of cores. And the amount of traffic in SF policy is normalized to its corresponding traffic in the baseline policy. In the SF policy, shared read traffic is what we expect to be reduced. This is because whenever someone issues a read request to a shared data, other sharers can fetch the data on the bus in advance. Therefore, the subsequent shared read misses will be decreased and the shared read traffic will also be diminished.

We can see that shared read traffic does decrease in almost all the workloads, and the reduction of the traffic almost all comes from the loss of the shared read traffic. However, there is also a reduction in private read traffic. In our conjecture, private read traffic should not get any benefits from our SF policy. In fact, the reduction of private read traffic can be regarded as a special case of the reduction of shared read traffic. For instance, if one shared data is going to be requested by two different processors. However, there is a pretty long interval between these two requests. The shared data may be evicted from the L1 cache in the processor which issues the former request as the latter request is issued, so the latter request would be treated as a private read request instead of a shared read request. In our SF policy, the latter request will become a hit in the SFB. Therefore, we diminish the amount of private read traffic. Besides, due to the unlimited size of SFBs, we can tolerate a much longer interval between requests. So more private read traffic can possibly be decreased in the experiment.

As we talked in Section 4.3, `canneal` and `lu` in the SS class have a trivial amount of sharing. These two workloads have only a few opportunities to take advantage of the SF policy. The reduction of the traffic mainly relies on the loss of private read traffic. We can also observe that `blackscholes` and `ferret` in the TS class have much less reduction than other workloads. As the example we present in Figure 5.2, once a shared data is modified by one processor, the next sharer who tries to access this data should issue a read request to get the modified data. In the meantime, other sharers can fetch the data on the bus in advance and start to acquire the benefits from the SF policy. Therefore, it needs at least three sharers to take advantage

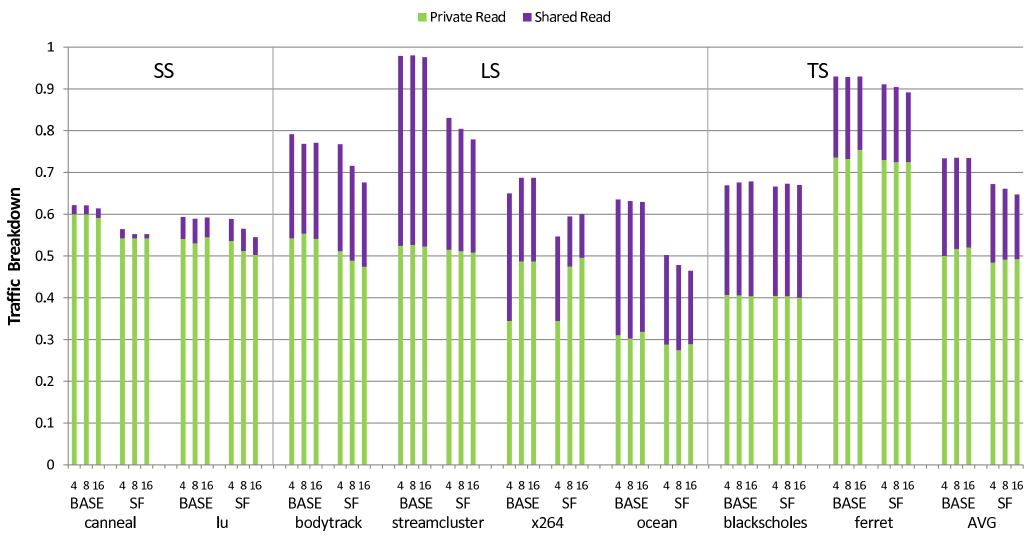

Figure 6.1: Traffic Breakdown between Baseline Policy and SF Policy in Different Number of Cores (PART1)

of the SF policy. However, most of the shared data are accessed between two sharers in the TS class. As a result, the workloads in the TS class can only gain few benefits in our approach. These benefits are derived from two situations. One is the trivial amount of the sharing behaviours with more than two sharers, and the other is the first time both the sharers try to load the shared data. The latter case is because that while one sharer fetches the data for the first time, the other sharer can also issue a fetch by the SF policy since it saw a read request for the shared data. The LS class presents a great improvement in the SF policy due to their large amounts of sharing behaviours. We can achieve a cache misses reduction of 8% in average and up to 20% in `streamcluster`.

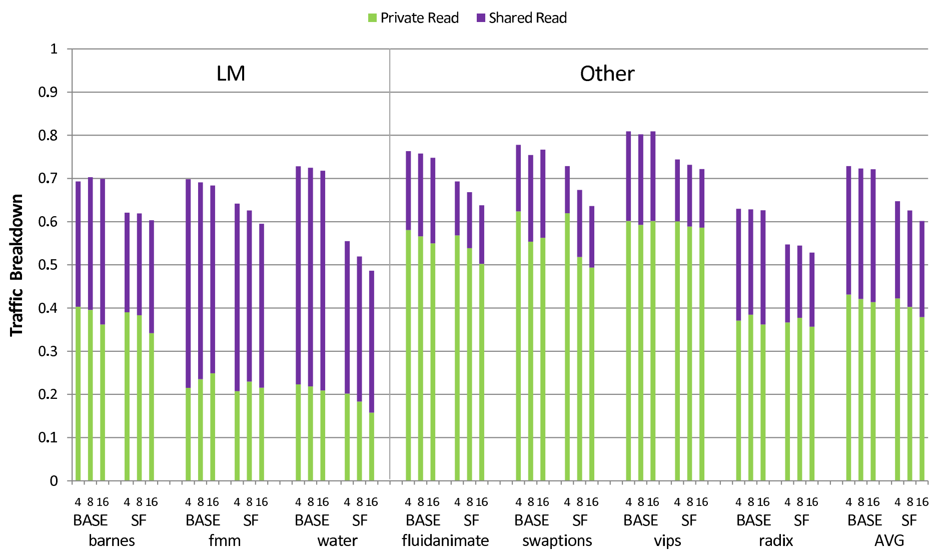

Figure 6.2: Traffic Breakdown between Baseline Policy and SF Policy in Different Number of Cores (PART2)

6.2 Number of Cores

In this experiment, we adjust the number of cores in the simulated machine and evaluate the effects of different workloads between the baseline policy and our SF policy in L1 cache miss rate and performance(i.e. CPI).

6.2.1 L1 Cache Miss Rate

In Figure 6.3 and Figure 6.4, we can find that almost all the workloads get a lower cache miss rate with a higher number of cores. Most programs exploit the data-level parallelism to implement the parallelization. Therefore, the size of working set for each thread gets smaller when more threads are

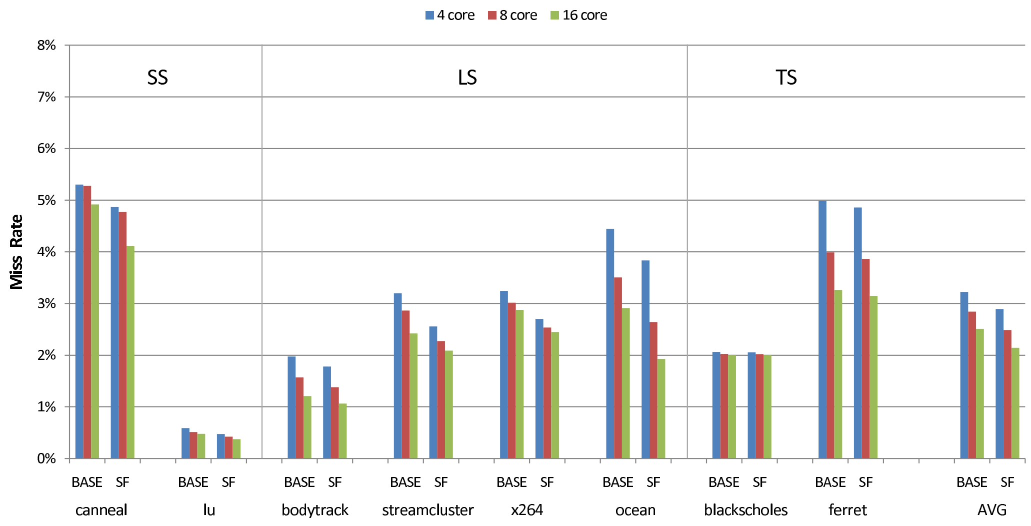

Figure 6.3: L1 Cache Miss Rate Comparison between Baseline Policy and SF Policy in Different Number of Cores (PART1)

divided in a higher number of cores. Since the L1 cache size for each processor remains the same, the cache miss rate decreases with the increasing of the number of cores.

When applying the SF policy, the improvements in the LS class are noticeable among all of the workloads. `radix` also shows a great amount of improvements in the SF policy. Although the amount of sharing in `radix` is less than the workloads in the LS class, its ratio of shared read traffic accounts for a large proportion of total sharing behaviours. We can see that it achieves about 1% reduction in L1 cache miss rate. Refer to the workloads in the LM class, the impact on their exceedingly low cache miss rate is slight even though they present lots of traffic reduction in Figure 6.2. Generally, the L1 cache miss rate presents a reduction of 0.4% in average.

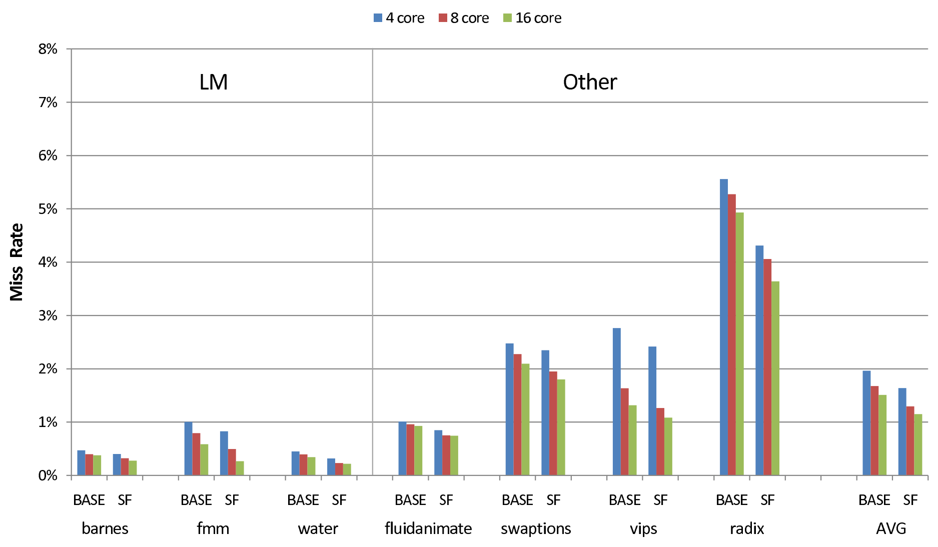

Figure 6.4: L1 Cache Miss Rate Comparison between Baseline Policy and SF Policy in Different Number of Cores (PART2)

6.2.2 Performance

In our expectation, the SF policy should gain more benefits from the higher sharing degree in a larger number of cores. However, we can observe that there is no identical trend with the increasing number of cores in Figure 6.5. The reason is that the cache miss rate is also decreased with the increasing number of cores as Figure 6.3 and Figure 6.4 present. The lower cache miss rate diminishes the demands for the data which we fetched in advance from the SF policy. As we discussed in Section 6.2.1, the LS class and `radix` present more improvements in performance(i.e. CPI) because of their larger amounts of shared read traffic. Due to the property of only two sharers, the TS class shows practically no improvements from the SF policy. We achieve a speedup up to about 1% and around 0.4% in average over baseline.

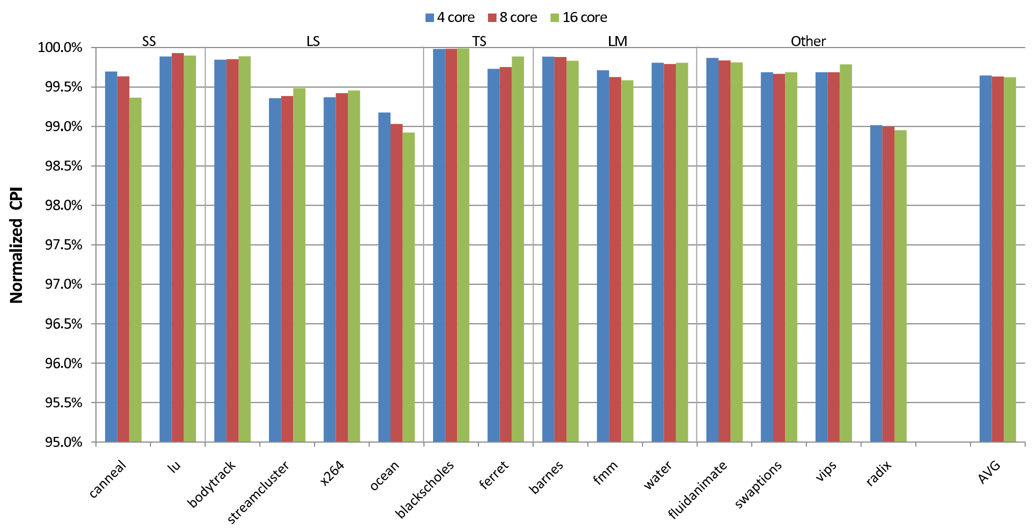

Figure 6.5: The Speedup of SF Policy with Different Core Number

6.3 L1 Cache Size

In this experiment, we adjust the L1 cache size in the simulated machine and evaluate the effects of different workloads between the baseline policy and our SF policy in L1 cache miss rate and performance(i.e. CPI).

6.3.1 L1 Cache Miss Rate

In Figure 6.6 and Figure 6.7, we can observe that as the L1 cache size increases, the potential of SF policy slightly lessens in almost all the workloads. Although we may meet more invalidated cache lines in a larger cache, the ratio of this condition to the total amounts of memory requests is still negligible. Since a smaller cache implies more cache misses, there are more

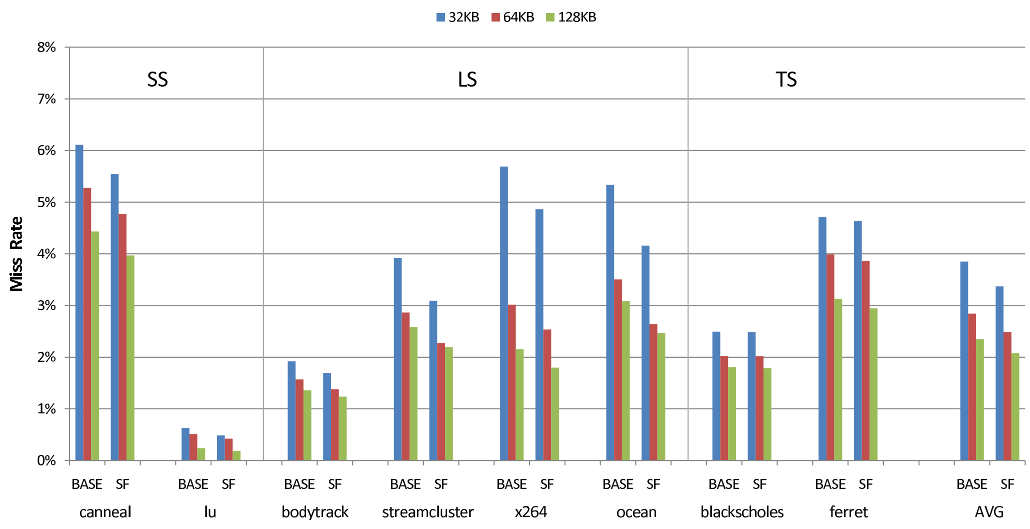

Figure 6.6: L1 Cache Miss Rate Comparison between Baseline Policy and SF Policy in Different Cache Size (PART1)

opportunities to gain benefits from our policy.

Because the TS class gives a extremely lower sharing degree than other workloads, our approach is unable to extract much benefits from this kind of property. `streamcluster`, `x264` and `ocean` in the LS class all present more than 0.8% reduction in cache miss rate from the SF policy due to their great sharing degree and the aggressive competition for cache resources. `radix` also presents up to about 1% cache miss rate reduction because of its great amounts of shared read misses as we mentioned in Section 6.2.1. According to Figure 4.8, we can find that the LM class shows a high sharing degree. However, owing to their low cache miss rate, the miss rate reduction isn't impressed in the experiment result even though they still decrease lots of miss requests. In a smaller cache size, we can achieve a reduction of 0.5% in average in L1 cache miss rate.

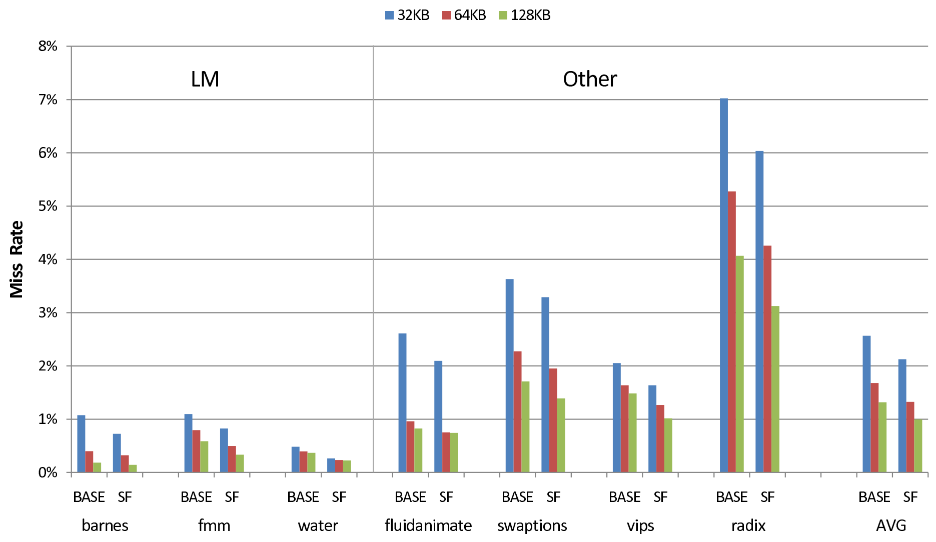

Figure 6.7: L1 Cache Miss Rate Comparison between Baseline Policy and SF Policy in Different Cache Size (PART2)

6.3.2 Performance

Figure 6.8 presents that the performance(i.e. CPI) just corresponds to the cache miss rate reduction showed in Figure 6.6 and Figure 6.7. As the same reason for cache miss rate reduction we discussed in Section 6.3.1, the performance(i.e. CPI) in a larger cache also exhibits a less improvement from our approach. The LS class and `radix` still perform a larger improvement than all the other workloads because they have lots of shared read misses. The workloads in the TS class have a similar performance(i.e. CPI) to the baseline policy. Since there is almost no sharing between more than three sharers in the TS class, it barely gets benefits from the SF policy. We can get an average of 0.4% speedup and up to about 1% speedup over baseline.

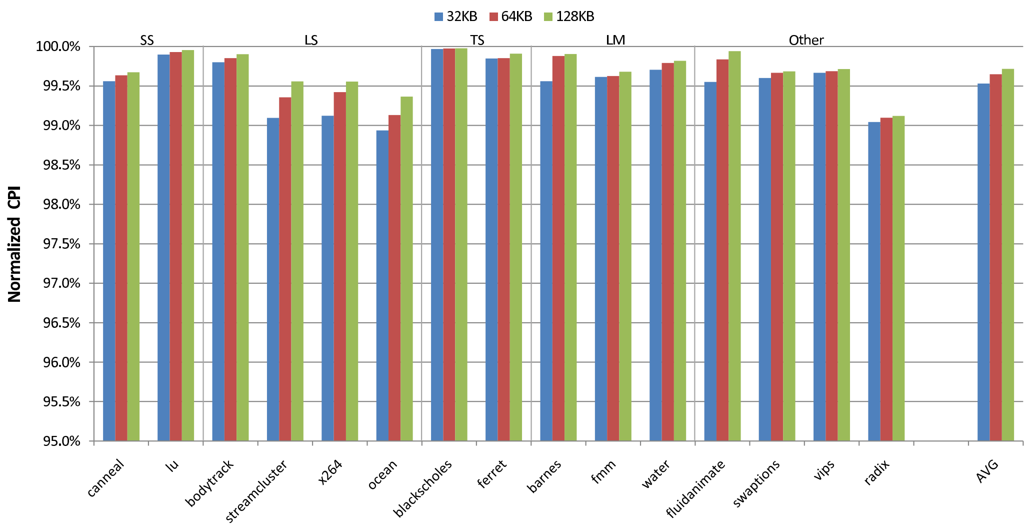

Figure 6.8: The Speedup of SF Policy with Different Cache Size

Chapter 7

Related Work

7.1 Workload Characterization

PARSEC and SPLASH-2 benchmark suites are widely used in many different researches for the multiprocessor systems. They both provide parallel programs for the evaluation of architectural ideas and tradeoffs. Bienia et al.[2] provide an overview of the PARSEC suite. PARSEC suite assembles a parallel program selection that is large and diverse enough to be sufficiently representative for scientific studies. This paper illustrates the behaviour of each workload and analyzes the parallelization, working sets, locality and traffic. A second version of PARSEC has been released in PARSEC 2.0[3]. SPLASH-2 is also a suite composed of multi-threaded applications. Woo et al.[18] expand and modify the original SPLASH programs to provide a broader coverage of applications and a better interaction with modern systems. They quantitatively characterize the SPLASH-2 programs in terms of fundamental properties like PARSEC[2]. This paper also provides some specific guidelines for pruning the space. Bienia et al.[1] analyze the PARSEC and SPLASH-2 suites for instruction mix, communication and memory behaviour on CMPs. It shows that SPLASH-2 and PARSEC are composed of programs with fundamentally different properties. Each divergence in the experiment comes from a distinct reason. In our work, we also try to characterize the benchmark suites with different parameters and observe the diverse behaviours of these workloads. Different from the above-mentioned papers, we try to evaluate the effect on the workloads when applying a software prefetching policy.

7.2 Hiding Data Access Latency

To hide the data access latency, predicting the following requests which are going to happen in the foreseeable future is one way to achieve. The prediction has been studied for a long time. Many researches are trying to propose a efficient way to make a effective prediction and incur less overhead in the meantime.

Sharing patterns are widely used as a hint to predict the subsequent behaviours. Mukherjee et al.[14] and Lai et al.[9] bring an idea inspired by a two-level branch predictor. They monitor the coherence activities and store these activities in a buffer. The buffer is a two-level buffer in each directory. The first level buffer, called Memory History Table (MHT), is indexed by the data block address and each entry contains the most recently incoming coherence activity and a pointer to the next level buffer. The second level buffer, called Pattern History Table (PHT), stores all observed sequences of coherence messages. Following the history information, the prediction can be made. Kaxiras et al.[8] use the instruction as an index instead of using the block address. Since code is much smaller than datasets, using instruction-based predictor can get a great deal of entry reduction and therefore needs fewer hardware resources. The instruction-based prediction examines the behaviours of load and store instructions in relation to coherence events and also keeps track of the history information in a buffer to make predictions. However, these papers we mentioned above only predict the next data access when meeting the current request. Wenisch et al.[17] explore that groups of shared addresses tend to be accessed together with the same order, and also recently accessed address streams are likely to recur. Based on their observation, this paper proposes a scheme to eliminate coherent read misses by streaming data to a processor. They use the miss history from recent sharers and move data to a subsequent sharer in advance of data requests. Somogyi et al.[16] exploit not only the temporal streaming but also the spatial streaming. This paper records and replays the temporal sequence of region accesses as well as uses spatial relationships within each region to dynamically reconstruct a predicted total miss order. All these papers keep track of a history information and issue prefetch requests based on the prediction. These properties cause extra bandwidth issues. However, there is no bandwidth overhead in the SF policy since our approach incurs no additional transmissions.

Since coherence protocol is a necessary to maintain the consistency of the data between processors, it incurs some extra coherence traffic. So to hide the data access latency, another way is to mask or even reduce those traffic. Huh et al.[7] have an observation that false sharing[6] and silent store[10][11] take a great fraction of the coherence misses. Those misses can be ignored because the cache has a correct data but the wrong coherence state. Therefore, they break the communication of a shared value into two constituent parts: One is the acquisition and use of the value, and the other is the communication of the coherence permissions that indicate the correctness of the value and thus the execution. The first part applies a speculative cache lookup and computation, and the second half uses the original coherence protocol to provide a mean for detecting a mis-speculation and recovering correctly from it. Cheng et al.[5] focus on improving the performance of applications that exhibit a producer-consumer sharing. They propose a novel directory delegation mechanism whereby the home node of a particular cache line of data can be delegated to another node. During the period in which the directory ownership is delegated, the home node forwards requests for the cache line to the delegated home node. Other nodes that learn of the delegation can send requests directly to the delegated node, bypassing the original directory as long as the delegation persists. Moreover, the mechanism is extended to enable the producer to speculatively forward newly written data to the nodes which are believed that the nodes are likely to consume it in the near future. Comparing to these two policies, the SF policy doesn't have to change the inherent coherence protocol. The only difference is to fetch the data during the regular snooping and the fetched data is in a shared state.

Chapter 8

Conclusions

In this thesis, we characterize the workloads in PARSEC and SPLASH-2 suites with different configurations. Lager cache size and larger line size both favor the cache miss rate. We also give a comparison of switching on or off the software prefetching scheme. Most applications benefit from the effect of prefetching. But the cache pollution issue should be taken care when a small cache size is applied. The sharing behaviours are remarkable in the multi-threaded programs. We observe that there is about 22% to 32% shared read traffic on the bus interconnection in both PARSEC and SPLASH-2 suites. It also implies that there are plenty of shared read misses.

Since there are a large amount of shared read misses in both PARSEC and SPLASH-2 suites, a *Snooping Fetch (SF)* policy is proposed to reduce these misses by making use of the shared data which is transmitted between processors in a snooping protocol. The SF policy is to fetch the data on the bus in advance once a read request is issued and the data is placed on the bus. Our approach achieves a reduction potential of 8% in average and up to 20% in traffic on the bus between the L1 and L2 caches over baseline, a reduction potential of 0.5% and up to 1.2% in L1 cache miss rate over baseline, and a potential of about 0.4% speedup in average and up to 1% speedup over baseline.

Bibliography

[1] C. Bienia, S. Kumar, and K. Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proceedings of the 2008 International Symposium on Workload Characterization*, September 2008.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[3] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[4] F. Black and M. S. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973.

[5] L. Cheng, J. B. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance*

Computer Architecture, pages 328–339, Washington, DC, USA, 2007. IEEE Computer Society.

[6] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. *SIGARCH Comput. Archit. News*, 21(2):88–97, 1993.

[7] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: making use of incoherence. *SIGARCH Comput. Archit. News*, 32(5):97–106, 2004.

[8] S. Kaxiras and J. R. Goodman. Improving cc-numa performance using instruction-based prediction. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 161, Washington, DC, USA, 1999. IEEE Computer Society.

[9] A.-C. Lai and B. Falsafi. Memory sharing predictor: the key to a speculative coherent dsm. *SIGARCH Comput. Archit. News*, 27(2):172–183, 1999.

[10] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 22–31, New York, NY, USA, 2000. ACM.

[11] K. M. Lepak and M. H. Lipasti. Temporally silent stores. *SIGARCH Comput. Archit. News*, 30(5):30–41, 2002.

[12] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

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

[14] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. *SIGARCH Comput. Archit. News*, 26(3):179–190, 1998.

[15] J. P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. Technical report, Stanford, CA, USA, 1991.

[16] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. *SIGARCH Comput. Archit. News*, 37(3):69–80, 2009.

[17] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 222–233, Washington, DC, USA, 2005. IEEE Computer Society.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.