UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Data Shepherding: Cache design for future large scale chips

Permalink
https://escholarship.org/uc/item/4bq2t8q5

Author
Jang, Ganghee

Publication Date
2016

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Data Shepherding: Cache design for future large scale chips

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Ganghee Jang

Dissertation Committee:
Professor Jean-Luc Gaudiot, Chair
Professor Alexander Veidenbaum
Professor Nader Bagherzadeh

2016

© 2016 Ganghee Jang

DEDICATION

To my wife, Jiyoung. Her courage led me in the right direction of this journey.
To my kids, Miru and Hechan. I felt re-energized whenever I saw them throughout the course of my study.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vii**

ACKNOWLEDGMENTS **viii**

CURRICULUM VITAE **ix**

ABSTRACT OF THE DISSERTATION **x**

1 Introduction **1**
1.1 Background and motivation 1
1.2 Problem definition 2
1.3 Goal of this work 3

2 Background Research **5**
2.1 System designs from the view point of design goals 6
2.1.1 Power wall problem and system design goals 6
2.1.2 System designs with dynamic change of goals 12
2.2 New technologies and system design goals 13
2.2.1 3D technologies and Non-Volatile Memory technologies 14
2.2.2 Reliability and design issues with other goals 18

3 Problem Definition **22**
3.1 Logical addressing mechanism to combine the multiple design goals 23
3.1.1 System design requirements and combining different goal driven designs 23
3.1.2 Merging different goal driven mechanisms and explicit resource addressing 26
3.2 Design tradeoffs 29
3.2.1 How extra resource indexing can handle design challenges on the last level shared cache 29
3.2.2 Design issues when integrating extra resource addressing on the last level shared cache 35

4 Data Shepherding Cache Description **40**
4.1 Overall Data Shepherding cache Design 41
4.2 Detailed mechanisms for Placement and Replacement 45
4.2.1 Retrieving bank mapping information and TLB cache design 45
4.2.2 Management of a mapping between a page and a bank 47
4.2.3 Description of Replacement 51
4.3 Page flush mechanism 56
4.3.1 Estimation of performance impact due to page flush 56
4.3.2 Page flush mechanisms and cache coherence protocol 60
4.4 Physical configuration of Data Shepherding cache 65
4.4.1 Overview of physical configuration 65
4.4.2 Discussion on the bank design 69

5 Simulation Configuration **72**
5.1 Overview of experiment setup 72
5.1.1 Goal of simulation setup 72
5.1.2 Comparison between Data Shepherding and Static NUCA cache setup 75
5.2 Implementation details on the simulator GEM5 78
5.2.1 Modeling of the Data Shepherding mechanisms in the TLB 78
5.2.2 Modeling a page flush 86
5.3 Resource Conflict Modeling 92
5.3.1 Overall the Resource Conflict Modeling design 92
5.3.2 The Resource Conflict Modeling and access timing 97

6 Results **101**
6.1 Analysis of physical characteristics 101
6.1.1 Details of the SNUCA cache 102
6.1.2 Comparison between the Data Shepherding and the SNUCA cache configuration 105
6.2 Performance analysis 114
6.2.1 Experiment setup 115
6.2.2 Experiment results and discussion 117

7 Conclusion **134**

8 Future Research **136**

Bibliography **140**

LIST OF FIGURES

Page

3.1 Example of future chip architecture. 30
3.2 Examples of benefits with the extra resource indexing is used. 31
3.3 Example of cache partitioning. 33

4.1 cache line placement of the Data Shepherding Cache. 44
4.2 Race between coherent page replacement and requests to memory system. . . 54
4.3 Overall flow of a cache line flush. 61
4.4 Start of a page flush at Bank 0 in the L2 cache. 64
4.5 End of a page flush at Bank 0 in the L2 cache. 66
4.6 The overall configuration of the last level shared cache of the Data Shepherding cache. 67

5.1 Overview of the Data Shepherding cache simulation setup 80
5.2 Flow of transactions from and to a sequencer. 84

6.1 Single bank access latency of the Data Shepherding cache and the SNUCA cache. 106
6.2 Latency of the longer link (between one for x-axis and the other for y-axis) of the Data Shepherding cache and the SNUCA cache. 107
6.3 Chip area comparison with ratio over the UCA cache. Each cache is sized as 128 MB. 110
6.4 Percentage of on-chip network overhead over total area. 111
6.5 Leakage power comparisons with sum of bank leakage power over the 128 MB UCA cache and the DS cache over the SNUCA cache. 112
6.6 Performance comparison between the DS and the SNUCA cache. 2 MB bank size, 128 MB in total. 118
6.7 The total counts of bank conflicts. 2 MB bank size, 128 MB in total. 119
6.8 The total counts of tag latency accesses. 2 MB bank size, 128 MB in total. . . 120
6.9 Performance comparison between the DS and the SNUCA cache. 2 MB bank size, 8 MB in total. 122
6.10 The number of page flushes. 2 MB bank size, 8 MB in total. 123
6.11 The total counts of bank conflicts. 2 MB bank size, 8 MB in total. 124
6.12 The total counts of tag latency accesses. 2 MB bank size, 8 MB in total. . . 125
6.13 Maximum, minimum and average page flush time comparison. 2 MB bank size, 8 MB in total. 126

6.14 Performance comparison between the DS and the SNUCA cache. 512 KB bank size, 8 MB in total. 128
6.15 The total counts of bank conflicts. 512 KB bank size, 8 MB in total. 129
6.16 The total counts of tag latency accesses. 512 KB bank size, 8 MB in total. . 130
6.17 The number of page flushes. 512 KB bank size, 8 MB in total. 131
6.18 Maximum, minimum and average page flush time comparison. 512 KB bank size, 8 MB in total. 132

LIST OF TABLES

Page

5.1 Extra coherence messages for the Data Shepherding simulation. 87
5.2 Extra coherent messages and events on the L1 cache. 89
5.3 Extra coherent messages and events on the L2 cache. 91
5.4 Coherent states of a L2 cache line and its access permission and location. . . 97

6.1 CACTI settings used for experiments. 105
6.2 Average hop counts on 2D mesh network with varying nodes. 108
6.3 Selected average access time of the Data Shepherding cache and the SNUCA when 4 GHz operation frequency is assumed. 109
6.4 GEM5 settings used for experiments. 114

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Prof. Jean-Luc Gaudiot. He gave me opportunity and support for this great journey.

Without the love, sacrifice and support from my parents, Dongsoo Jang and Soonja Lee, I could not finish this journey. I must include my mother-in-law, Yuim Lee (my wife's mother) to this big thank-you list. Without her sacrifice and support, I could not finish my study especially after my parents' business went to bad.

During my PhD study, I met many visitors to the lab, and the conversation with them helped me a lot. Especially, I want to say special thank-you to Prof. Eui-Young Chung and Prof. Seong-Won Lee for their advices on the generic approach toward research.

I am really lucky to have good seniors and friends as being a part of the PASCAL lab. Alex Chu and Howard Wong were great office mates, and also we made many hang-outs which saved me from the research stress.

Between the people I met and become friends outside the lab, I feel really appreciated to David Keilson for all his advice on life in US. He is such a great mentor and citizen.

Also, I learned a lot from my kids, Miru and Hechan. They broadened my views on the value of education. I cannot help mentioning about my electric guitar. It gave me a new insight on what a good invention should be - a guitar has made people happy (including me) without hurting anyone.

CURRICULUM VITAE

Ganghee Jang

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering **2016**
University of California, Irvine *Irvine, California*

Master of Science in Electrical and Computer Engineering **2006**
University of Florida *Gainesville, Florida*

Bachelor of Engineering in Electronics Engineering **2003**
Korea University *Seoul, South Korea*

EXPERIENCE

Graduate Student Researcher **2010–2015**
University of California, Irvine *Irvine, California*

Teaching Experience as Grader **2011–2013**
University of California, Irvine *Irvine, California*

Network System Engineer **2003–2004**
Korea Telecom *Seoul, South Korea*

SERVICIES

Reviewer **2010, 2012, 2014, 2015**
SBAC-PAD

Reviewer **2012**
TPDS

Reviewer **2011, 2012**
DFM

ABSTRACT OF THE DISSERTATION

Data Shepherding: Cache design for future large scale chips

By

Ganghee Jang

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2016

Professor Jean-Luc Gaudiot, Chair

The issue of the power wall has had a drastic impact on many aspects of system design. Even though frequency scaling is limited because Dennard scaling stopped a few years ago, new technologies and design trends such as 3D stacking and heterogeneous multi-core architectures still offer us opportunities to integrate more logic to increase performance. However, these new trends have increased design complexity significantly, especially in terms of satisfying various system requirements and system goals: power efficiency must be handled carefully while scaled resources must be managed to improve performance. Sometimes system scaling itself conflict with system goals. Increased chip resources and core counts make fair resource sharing more notable issues, and the increased number of heterogeneous architecture makes programming more challenging by requiring more programming styles (e.g., stream programming, SPMD style programming), API, etc.

We suggest here that multiple design goal issues could be handled more efficiently via explicit resource addressing. This idea is applied to the shared last level cache design and we call this approach the Data Shepherding cache. We show how the Data Shepherding cache design solves some of the design issues of integrating mechanisms from multiple goals especially when we consider scaling in capacity and performance. The properties of the Data Shepherding cache are evaluated in two ways with open source tools. CACTI from HP.com simulates cache

models to obtain cache parameters such as access time, power consumption, etc. Further analysis with the model parameters shows that the Data Shepherding cache has 10% smaller area footage, which leads to a 7% leakage power reduction over the Static NUCA cache. We analyze performance using the GEM5 simulator from gem5.org which models system architecture. The results show a 12.6% performance improvement with selected Spec2000 benchmarks with 2MB bank, 128 MB total shared cache size.

Chapter 1

Introduction

1.1 Background and motivation

Recently, we have observed a lot of challenges which lead to turns of technology trends. The issue of the power wall has had a drastic impact on many aspects of system design as we have seen the rise of multi-core design on the market. Frequency scaling also slowed down because Dennard scaling [36] is not effective any more. However, new technologies and design trends such as 3D stacking [19, 26] and Non-volatile memory technologies [62, 30] still give us room for a steady scaling in number of logics on a chip. Interestingly, new design trend such as heterogeneous architectures and Application specific Processing Units (APUs) let us transform the benefits of scaling in count into improving performance or power efficiency.

By the way, this new design trends come from utilization of extra logics. In other words, performance/power efficiency improvements are achieved at the cost of the design complexity. This induces further implications. Significant re-design may require to improve performance of the scaled version of an existing design. Mechanisms which are attached to the major components for extra functionalities also require significant change. The extra functionalities

are to meet the design requirements, or goals such as power efficiency, performance efficiency, resource sharing management, etc. For example, power control mechanisms on the shared cache design [37, 74, 41, 72, 57, 15] have suggested to control the amount of leakage power because the amount of area for the last level shared cache takes significant portion of chip area. As we will look at in Chapter 2, some of them is not easy to apply to the much larger caches than the sizes of the cache which the authors worked on. There are some which would be tough to be used together with other goal driven mechanisms. These issue of scaling and goal driven designs will be discussed in detail in Chapter 2.

1.2 Problem definition

A good example of goal driven designs are found in the cores of multi-core chips. The power and performance efficiency of each core can be controlled via Dynamic Voltage Frequency Scaling (DVFS) mechanism on each core. Global policy to achieve power consumption targets or performance targets can be optimized through DVFS control on each core. This is possible because each core is addressable to the global policy management. This explicit addressing is not popularly used with the last level shared cache design because explicit addressing generally had been linked with programmable cache memory. Some examples can be found in the domain of embedded system design where high degree of optimization is required.

The reason for the weak popularity is because of design approach on the cache memory. The cache memory has been considered as a buffer memory, so search has been regarded as a primary method to find a desired content in the cache memory. However, as we scale a chip size, more design concerns would be added, especially, on the last level shared cache. As discussed in [9, 92], the number of pin for data transfer might fail to catch up the required bandwidth due to the cost. 3D stacking technology and the use of heterogeneous

architecture design can increase on-chip network complexity as dimensions and node counts increase. These issues require high degree of specialization on the last level shared cache [68, 47, 105].

We can find a good design example for these design issues in the domain of building architectures. Modern skyscrapers have managed similar design issues very efficiently. They absorb on and off building traffic by having multiple functional areas of living area, business area, shopping area, etc. as we can see some design approach of integrating the main memory with the CPU[105]. Traffic in the building is efficiently managed with the more complex elevator system, which is related to the performance efficiency. The solutions are possible because placement such as elevator stops and office locations can be adjusted. Actually, the adjustment can be done with the floor numbers. The floor numbers give efficient an abstraction to add multiple functionalities.

Many design issues with the cache design could be solved if similar design approach to the addressing with floor numbers of skyscrapers would be applied. We call it explicit resource addressing design strategy, and in Chapter 3, we will explore how this can solve the raised design issues. Also, in Section 3.2.2, expected challenges will be discussed when we apply the explicit resource addressing on the last level shared cache design.

1.3 Goal of this work

As identified challenges can be solved efficiently with explicit resource addressing on the last level shared cache design, now we have a question of whether we can make a scalable cache design with explicit resource addressing. Because adding explicit resource addressing has comparable design issues to adding logical addressing, there are design issues such as mapping table management, when and how to make mappings. By the way, as we can

see 128 MB of shared last level cache on the market[5], careful consideration of mapping granularity is a key to manage implementation overhead. However, bigger granularity has its own drawbacks. Bigger granularity would increase internal fragmentation and false sharing. Also, if the granularity size is the same with the granularity of the virtual memory system, we must consider performance interference.

We selected a page size as the mapping granularity, and tried to design so that it should work for scaling in size. The purpose of explicit resource addressing is to resolve issues from the future scaled system, so it is important to imbue good features for the scalability. Detailed design of the Data Shepherding cache is discussed in Chapter 4. We assign a separate chapter (Chapter 5) to explain implementation details on the Simulator, GEM5 [25].

Because of increased granularity, the Data Shepherding cache attains less hardware overhead, but gets performance overhead of page flushes. In Chapter 6, we will evaluate the Data Shepherding cache in two ways. We will estimate the benefits from the reduced hardware overhead. The hardware parameters for the estimation is modeled from CACTI tool[78]. Then, detailed performance evaluation is made with Simulator, GEM5 to examine how the overall performance is influenced from the reduced hardware overhead and performance overhead. Observations from the results will be summarized in Chapter 7. Remained and extended works planned for the future is listed in Chapter 8.

Chapter 2

Background Research

The new technology and design trends are giving us interesting design challenges. The new technology and new design trends are not just scaled versions of previous designs, but sometimes drive into the direction of significant new designs. For example, recently we observed the very important technology turn because of "power wall" and saw the rise of multi-/many-core designs. Power wall is not the only challenge which has changed the design trend. We will look into these new design trends from a view point of design goals, and discuss the efficiency of the increased design complexity.

Actually, this chapter is to lead why the explicit resource indexing matters as a interface to implement mechanisms to achieve various design goals. So, the readers who want to read directly from the Data Shepherding cache design may skip this chapter.

2.1 System designs from the view point of design goals

2.1.1 Power wall problem and system design goals

Embedded systems were considered more specific design goal driven as we can see many application specific designs such as realtime systems, ATM systems, etc. On the while, systems such as desktop PC systems fell on the generic system category. However, "power wall" steered huge turn in the design trend. We already saw that major performance matric for the consumer desktop system have moved to the system throughput from the system frequency because robust frequency scaling becomes impossible. One good example is that Intel canceled Tejas CPU design around 2004 [40]. The chip was originally planned to run up to 7 GHz, but power dissipation of 2.7 GHz operation was 150 watts, so failed to be scaled up.

There are other important design challenges which have changed the design trend significantly. Actually, the advent of smart and wearable technology converts the "generic" system into more goal driven designs. The cortex ARM based processors [61] popularly used in the smart phones and tablet PCs can handle various types of workloads. It is a good example of goal driven (in this case, power efficiency) application of "generic" CPU. Apple Inc.'s new tablet with embedded processor was announced that its performance was par with the entry level laptop PC [1]. This power efficiency driven design also can be seen even in the domain of super computers. There is a good example in the article of "Exascale Computing Technology Challenges" [87]. US Department of Energy sets limits of power consumption of exascale system to 20MW considering Total Cost of Ownership (TCO). However, the standard memory technology road map estimated that just total access to DRAM would contribute roughly 48MW, which contributes one of the important challenges. To meet the requirements, the system design must be goal-driven - the design goal of power efficiency.

The power wall issue steers our design approach into the new direction. As we have discussed with the exascale challenge, and the advent of many-/multi-cores, the solution would not be a summation of local solutions such as a design with power efficient devices. Rather, significant design changes were required, for example, multi-/many-core architecture instead of single high performance core.

It also changed our view on the system design. Before the turn, a system stayed in more static status and a designer focused more on generic features. For example, a design for a generic desktop processor focused on the performance improvement of common workloads or the most dominant workloads. The others depend on the coding skills of a programmer, or better software algorithms. Of course, the days when frequency scaling was feasible, the workloads without the help from the special hardware support also got performance improvement from the frequency scaling. However, after the turn, frequency scaling is not robust any more when design a new product, chip designers depend more on the application specific circuitries. As a result, there can be many under utilized system resources with a specific workload, or there can be some system resources which are heavily utilized. Systems become more dynamically configured over time, and designers welcome specialized components.

We will discuss on the first with the example of cache memory designs. Especially, we will focus on the problems when a special goal driven component is used with other component or other system goals. The second will be discussed more in the next sub-section with design examples of dynamic system configurations.

Cache Designs to handle leakage power

Power dissipation consists of dynamic power, short-circuit power, and static power dissipation. Dynamic power consumption had been a major challenge because of its larger share of the overall power budget. However, the growing importance of low power design and current

sub-micron technology makes static power dissipation more significant problem. According to ITRS [52], leakage power is increasing exponentially, so it was expected to dominate overall power consumption. However, thankfully, new technologies such as high-K dielectric [71]and Tri-gate transistor design [11] changed the expected trajectory, so we did not see the exponential increase of leakage power yet.

Anyway, it is still obvious that the leakage power is one of the most important citizen to design power efficient components, so many designs had been suggested to control the leakage under power budget. The most efficient way of reducing the leakage current is shutting-down a component - the technique called power gating. However, the delay to power up a component is very expensive if we want to apply to the latency sensitive component such as a cache memory. So, to use power the gating technique at the cache memory design, we have strong credibility on some part of the cache will not be used at least for some specific amount of time which can compensate the overhead. With weaker credibility, it would be safer to use operation mode with reduced supply voltage. Careful experiment on the choice of supply voltage can find good balance between performance loss and decent deduction on the leakage current.

Let's look at examples of the latter cases first. As a extreme example, we can lower supply voltage as low as sub-threshold level as in [37, 74]. Sub-threshold device designs have benefits on the power consumption and area usage, but at the cost of some performance loss. In case of computing components, we can compensate performance loss via parallelism because of small implementation footprint. However, in case of latency sensitive memory system such as cache memory system, we cannot adapt the idea of throughput.

Drowsy cache [41] took interesting twist. The authors set an operation mode called "drowsy mode" which was selected to have couple of cycles to wake up the component. When not in use, a cache line goes into the drowsy mode, and it is waken up when accessed. When in the mode of operation, supply voltage is increased - so in case of successive accesses, there

will be no overhead. Slumberous cache [72] did more refinement with this idea with more operation modes. A cache line goes into more deeper sleep (lower operation voltage) as one is moved into more least recently used while becomes more sober with successive hits.

Now, it is time to consider the power gating. Careful consideration must be given to select a part of cache to turn off because turn-off period must be long enough to compensate turn-on and turn-off latencies. Cache Decay [57] takes strategy of turning off cache lines using counters. Counters are increased periodically, and a cache line is turned off when a counter is saturated. A decision to turn off is made statically, so sensitive to counter increase interval, cache size, etc.

When we select victims to be turned off dynamically, the most probable choice will be from the bottom of the LRU stack in a set [67]. (Actually, the above Slumberous cache utilized this characteristic.) Dynamic Non-Uniform Cache Access (DNUCA) cache is in a good configuration which we can make use of the LRU stack. Each cache line in a set spreads over different banks, and makes transitions closer to the core every time it gets accessed. Naturally, the banks forms array of LRU stacks - the closer to the cores, the more recently is a cache line used. So, the farthest banks become candidates to be turned off. This idea is implemented and experimented in [15].

This part of writing will be never complete without the discussion on the dynamic power. The characteristics of local and share last level cache are quite different from the view point of power control. The local cache is accessed very frequently, while the last level shared cache gets much less accesses than the local caches. The local cache tends to be smaller, while the last level cache is much bigger. So, dynamic power control have more visible effects on the local cache memory, while the leakage power control becomes more important on the last level cache. Dynamic power efficiency can be achieved from two directions. First, we can explore chances for the power saving in the cache module. In case of a set associative cache,

all the lines in a set are read out, and then one is selected. Power saving is possible if we can predict the exact match. This idea is called way-prediction and explored in [83, 18].

Another chances of saving dynamic power can be found from outside of a cache module. For example, in the inclusive cache memory, cache miss in the upper layer of the cache memory system cascades into the next lower layer of the cache memory system. If a hit/miss can be resolved early, then we can save extra power to search for a cache line. Early cache miss decision is discussed in [69].

Design issues with other goals

As we discussed briefly above, the power wall drove the popularity multi-/many-core architectures to compensate performance loss from the weakened frequency scaling. The focus of design trend has been changed from improving a single thread performance to improving overall system throughput. This trend change caused unexpected conflict of design goals. The multi-/many-core system runs multiple hardware threads or sometimes multiple processes concurrently, so compared to the time sharing of a single core system, the importance of resource sharing management spiked up in the system design.

The importance and designs for the fair sharing of the last level shared cache had been discussed by many at [94, 84, 86, 106]. Dynamic partitioning of shared cache memory [94] is one of the earliest approach to the cache partitioning in the multi-core system. It pointed out the possible cache pollution by a thread which could cause higher miss rates for other threads because of a single LRU scheme used in a set. Mechanisms to gather and keep marginal gain of additional cache ways for each partition are designed to reach out optimal point between partitions.

More refined approaches [84, 106] employed stack algorithm [67]. Utility based cache partitioning [84] discussed more efficient monitor design for stack algorithm. Dynamic set

sampling method is developed to reduce implementation overhead which is huge drawback to the stack algorithm. PIPP [106] controls cache partitioning by managing insertion point of a new cache line in a LRU stack of a set. The authors pointed out that the cause for the possible cache pollution would be the insertion point of a cache line when hit or newly assigned to the cache. Cache pollution is weakened by limiting the insertion point of a thread which causes cache pollution. However, cache ways are scaling slower then the number of cores, this methods are liable with cache pollution at the tails of a LRU stack. Vantage [86] took interesting twist on the cache partitioning. To solve weak scaling of cache ways for the cache partitioning, the authors exploited hashing to make a set over-grown. When a partition needs to be grown or reduced, a cache line is borrowed from and to unallocated area.

As we discussed the relationship between the lower positions in the LRU stack and the candidates to be turned off in [15], it may mislead us that the many of fair cache partitioning mechanisms are friendly to the addition of the power gating mechanisms. Unfortunately, complexity of design increases rapidly if we combine them without significant modifications. For example, mechanisms such as the PIPP [106] and the vantage [86] increase activities from LRU stack tail pollution or borrowing from unallocated area. This will decrease the average expectancy of credible time for turning off.

Even mechanisms without special algorithms to improve efficiency of partitioning management, there are still drawbacks of huge implementation overheads. Power control with multiple sleep modes [72, 57] added extra states of a cache line, so as the cache partitioning mechanisms. Also, both work on the cache block granularity, we would end up augmenting significant circuitry onto both the cache memory controller and tag arrays.

So, I. Kotera et.al [59] decided to increase control granularity to decrease implementation overheads. They actually controls the cache partitioning and the power management with the granularity of ways - by pinning the locations strictly in the set where as most of other

cache partitioning mechanisms used LRU information (to figure out the position in the LRU stack). The reason for the latter is to relax placement of cache lines in the partition. If we strictly enforce the placement, cache blocks may be flushed out into the main memory every time partitions are adjusted, which would be overhead on the performance especially when this mechanism would work generic OS with context switches. Also, the degree of set associativity is not scaling strongly compared to the number of cores, so the chances of implementing this scheme onto a system with more cores will be limited.

2.1.2 System designs with dynamic change of goals

The system designs we have discussed so far in this chapter are to achieve static design goals. In other words, the mechanisms to achieve the design goals work all the time until the system is turned off. For example, fair cache mechanisms run all the time towards the best system task throughput. Power control mechanisms always try to reduce leakage power consumption while keeping the performance loss towards possible minimum. However, the implication from a different direction of the power wall actually drove more diversity in the trend of architecture designs.

The dark silicon design era [97] is to stress out a period after another technology turn, "utilization wall". While power budget increases quadratically due to leakage power, utilization of a chip would be limited to the reverse of power budget as scaling of transistor technology. In other words, it would become exponentially difficult to power a chip with the same size with every new transistor technology scaling. The authors listed new design trends to overcome the utilization wall. The examples are Near Thresh Hold Voltage (NTHV) technologies, Dynamic Voltage Frequency Scaling (DVFS), etc. The DVFS technique is used to increase performance of a performance sensitive thread by increasing its supply voltage while other cores get less supply voltage. By doing so, a system would achieve better throughput

while keeping the power budget of a system. Interesting twist of this scheme is to let a DVFS system sprint until a system hits maximum cooling capacity. With repeated sprint and then slow-down, a system can achieve better performance. The power management with the DVFS scheme can have multiple configurations for different performance goals and can be changed by a programmer or OS.

Other example of dynamic goal changes can be found in the HPC design domain. Researchers [44, 99] discussed the possibility of exascale system by 2018 while suppressing power consumption around 20 MW range. The power budget is a result of many design goal conflicts - cooling, operation cost, etc. Because cooling is an important design challenge [14], as the case of the DVFS implementations we discussed, HPC system can be operated with multiple goals. We can intentionally run under-loaded nodes in a cooler state to save power while the nodes would be recovering their states into normal operating states quick when loaded fuller.

2.2 New technologies and system design goals

In this section, we will look at other technology trends and related design together with design issues with design goals. While Moore's law has proven robust over the best part of the last 30 years, there are indications that it is abating and that the exponential increase in the number of transistors per chip may come to an end within the next 10 years [98]. However, new technologies are on the horizon which may compensate for this trend. Indeed, 3D or 2.5D stacking technology [19, 26], and some new non-volatile memories [62, 95] are some of the most promising examples.

2.2.1 3D technologies and Non-Volatile Memory technologies

Overview of stacking technologies

Design with 3D technologies is being approached from many directions. The vertically stacked gate approach reduces the short channel effects very efficiently, which means a significant decrease in leakage power and faster switching. Stacked gate (Intel's tri-gate technology [60]) enhances power efficiency and performance at the same time. This technology has important applications at the architecture level. For example, consider the design of a server processor where memory bandwidth is known to be a limiting factor, which means that designers need to include comparatively large caches. Recent research [48] suggests that stacking huge L3 caches on a single chip will be a tremendous way to enhance performance and reduce power consumption, all without a significant increase in space.

The key benefit of 3D technology at the architectural level comes from the fact that heterogeneous processes or technologies can be integrated on the same chip in the different layers. For example, DRAM or SRAM layers can be stacked on top of a processor layer or vice versa. As a result, we can integrate more transistors within the given area by stacking them vertically. Another benefit comes from the TSV (Through-Silicon Via) technology which can be used as a support for high speed transport for data. Elpida [3] and Xilinx [8] have already utilized TSV to scale the capacity of their devices by connecting multiple modules on the same substrate via TSV. More specifically, these commercially available products [2, 26] are built on the same substrate and are good examples of 2.5D technology.

We expect that large scale system design will potentially benefit from these architectural characteristics of these new technologies. First, multi-layered integration will enable the integration of more computing resources in a single chip. Indeed, increasing the computing density is a crucial factor when we consider the expected scale of future computers. Second,

the ability to include multi-layered heterogeneity (3D) or heterogeneity on the same substrate (2.5D) will give us a better chance to render a chip more application-friendly by integrating a variety of application-specific accelerators. As we discussed in the last section, this feature will give us room for power efficiency, too. Turning off unused accelerators and selecting ones with the better power and performance efficiency should enable us to achieve increased degrees of node level performance and power efficiency.

Design issues with other goals

Of course, system design with 3D and 2.5D will not come without some downside. Indeed, packing more computing devices on a single chip will make efficient cooling more difficult, especially with 3D technology. Higher degrees of heterogeneity will add much more complexity to the system management because each node in the heterogeneous system has more variety in the power consumption and computation capacity.

Let's estimate the complexity with a model of a chip with four multi-cores and a GPGPU. Unlike the typical CPU, a GPU is specialized in data-parallel computations where hundreds of threads can simultaneously execute the same set of instructions on independent pieces of data. To accomplish this, most of the on-chip area of a GPGPU is dedicated to data paths, thus minimizing the size of the local caches and complex control logics. GPGPUs are power efficient from the view point of power density per computation, but the overall power consumption of a GPGPU unit is quite visible. The power consumption of AMD APU like model used in the work [103] is that the maximum power consumptions for each device are 80W for CPU, and 63W for GPGPU, but the total must be kept under 100W. Each section is subdivided into power control domains, for example, each core becomes a domain in the CPU side. Each domain works with variable power supply voltages for power control. So, we can expect the combinations would grow faster with the increase of heterogeneous

components in the chip or in the system. Performance boost from the scaling heterogeneous architectures would face challenges again from the design goal of power efficiency.

Steady scaling of transistor counts brings another challenge. Pin count scaling would not catch up the bandwidth required by 2020 without significant design efforts [92]. Design efforts such as on-chip memory controller, in-chip voltage regulators, etc. had been a help to suppress trajectory of scaling. Additional architectural design effort such as compression on the traffic between a CPU and the main memory [9] can be a help.

Block data transfer is also helpful for the bandwidth issue of a chip due to its high utilization of a link. On top of that, high link utilization sometimes can be lead to better performance from the high utilization of computing cores. This is especially true with stream programming, for example. IBM Cell [54] chip employed this design approach. It has one power processor core and 8 Synergetic Processing Elements(SPE) which is smaller than the power core. Each SPE has local memory, and is connected via ring based network controlled by Direct Memory Access (DMA) controller which resides inside each processing unit.

However, performance improvements come at a cost: their own local memory must be tailored for the kind of access they require and must be explicitly programmed [34]. Data must be divided into tiles, and the tiles must be assigned into input and output channels of a worker (processing unit) by a programmer. A programmer must understand memory system details precisely. The problem does not stop here. Because of the hardware details coded at the user level, the codes are not easily portable to other systems. Programmability becomes another challenge and system design goal.

Non-Volatile Memory technologies and design goals

Aside from 3D and 2.5D technologies, Non-Volatile Memory (NVM) technologies open up new system design opportunities as well. Power efficiency of memory system becomes crucial

because the portion of memory system in overall power budget is growing fast over time. For example, as discussed in [87], the power cost of memory operations will be the largest share in the overall power usage of exascale systems. NVM technologies have obvious edge on power consumption and area over volatile memory technologies. However, because of slower access time, applications of NVM once stayed in the limited design area.

The challenge from the power wall started to be considered into more performance sensitive designs. In [62], the authors discussed the designs for Phase-Change memories to be substituted for the main memory. Of course, the substitution could be done at the cost of considerable design efforts such as row buffering, row caching, wear reduction, wear leveling, etc., but the cost is acceptable to overcome the power wall and to keep scaling of the size of the main memory.

Researchers broadened the application of NVM even more. There are even some ideas such as building every memory layer into a single NVM layer [12]. With single address space, application states can be always kept as running because a binary can be run on the single layer of memory system. The implication of the design will be huge especially with OS design, so will require even more design efforts.

NVM technologies are currently entering into its next level. New technologies such as 3D NAND device technologies, and Intel 3D Xpoint [30] are expected to enhance access time and are efficiency significantly, they are more ready to replace the main memory technology. Near Memory data service [38] expected new design trends from these new technologies. Better performance means more chances for unified shared memory design, and reduced needs for the caching. This property may give a Processor In Memory (PIM) architecture a new life. Still, there would be performance disadvantage on the cross partition accesses, but this disadvantage would be mitigated with applications with lots of data parallelism. PIM architecture can be used for some models of nero-networks, so its application would broaden into the area of machine learning.

2.2.2 Reliability and design issues with other goals

We will look at the reliability issues especially with the two design issues of parameter variation and transient faults. Parameter variations affect the operation of a integrated circuit in various ways after fabrication. These can be categorized into several types: parametric variations, temporal variations, power and process variations, and cause variations in the timing, power, stability, and quality specifications, and eventually reduce yield rates and expected lifetime of a chip [43]. Parameter variations can occur during fabrication and its effects are permanent. Alternatively, transient faults cause errors such as bit flips due to radiation. Overall, they result in failure in achieving the performance desired by the user.

Parameter Variation and Reliability

Parameter variation is expected to cause even more challenging problems with verification and fabrication yield in deep sub-micron technology. However, efforts to enhance fabrication yield under parameter variation have been underway for some time already: for instance, vendors have sold chips with different cache memory size by shutting off a bank which has been determined, at verification time, to be faulty. Also, chips of varying quality from the same design are being commercialized for marginal profit. Generally, the quality differences are in their power consumption rating, their operation frequency, their operation voltage, or their number of cores.

Due to the continued exponential increase in the amount of available logic on a chip, the impact of parameter variation will become increasingly significant. In [10], the verification cost and the yield will take up a significant portion of the overall cost, which may even slow down the scaling of transistors in the future.

Thus, to compensate for the verification costs and increase the chip yield rate, various approaches have been explored. One possibility is to make a design more variation-tolerant at each design layer. The key idea would thus be to use the inherently available redundancy and some additional control logic to compensate to some degree for parameter variation or even permanent failures in the system logic [64] or the memory cells [76]. Mostly, we can expect that chips after verification are "good" chips. The other approach is, in a sense, bolder: chips with some internal faults can be sold but they will be provided with a more sophisticated architectural support to correct for incorrect execution results [10].

However, these solutions remain somewhat narrow-focused: while they can guarantee execution correctness, they do not take their impact on the overall system into account. For example, check-pointing and recovery can consume a significant number of clock cycles, and together with synchronization in large scale systems, it is not easy to evaluate the impact on the overall system performance. Indeed, automatic control mechanisms to compensate parameter variations often lack a system-level point of view, which sometimes means that other system design goals are ignored. For example, some techniques to compensate the variation of V_{TH} can lead to more variation in leakage current [24]. In other words, uniform performance is achieved at the cost of irregular power consumption.

Transient Faults and Reliability

Even a chip which has successfully passed the verification process can still exhibit some kinds of failures, especially some which are transient. Typical transient faults in CMOS technology-based chips are dielectrical breakdown [81], hot-carrier induced degradation [29], radiation-induced transient faults [63, 104], *etc.*

Generally, different techniques have been developed in control logic and memory cells in order to detect and correct the errors caused by the issues mentioned above. In the case

of control logic, redundant execution [88] and check-pointing [90] are some of the most popular methods. However, the cost for additional hardware or re-execution is significant on a chip with more concurrent execution contexts. However, both methods require significant numbers of transistors to add redundant logic or speculative storage. Both techniques require additional cycles to decide that the execution results are acceptable. In the case of check-pointing, it will take additional time to recover the previous state and re-execute. Redundant execution suffers from high hardware overhead if the control path is complex, and it causes low utilization of redundant logic. This issue is expected to become a particularly important issue with multi-core architecture design. Compiler generated signatures can be helpful to reduce overhead as discussed in [63].

On the other hand, for the memory cells, hardware-based methods such as an ECC memory, utilizing redundant cells, *etc.* are a better choice. Under more faulty situations, other hardware-based approaches are favored. For example, a micro controller can be embedded inside a NAND Flash memory-based SSD to handle more complicated control requirements such as tracking faulty cells and remapping their use with healthy cells [65]. Wear-leveling is done with the complex control logics to extend the life-cycle [31].

Even with continuing improvements in operation frequency and feature size, new technologies like 3D-stacking are offering yet another approach to continue increasing the density of transistor integration. For instance, the Mean-Time-To-Failure (MTTF) of multiple hard disks [49] will be much smaller than that of a single hard disk. Further, the issue of reliability raises even more complicated design issues: in [79], the mere possibility of degradation of a system could actually limit the scalability of a large scale multi-processor systems. This is actually an excellent example how system design can be deeply entangled with the multiple design goals of performance and reliability.

In [70], to address the problem of conflicting goals in the design of a system, the concept of performability was suggested and discussed. One key idea behind performability is that per-

formance is degradable, and must thus be considered together with reliability. The challenge the author found was the discrepancy between performance metrics and reliability metrics, and thus suggested a specific model based system evaluation to fill the gap. However, while the idea was modeled and developed with communication or network system, it has some limitations to be applied to the design of future systems. Many models come from the stochastic random process model, so inherently assumes each component in the system is homogeneous. Besides, even some components like FPGA are mutable. Furthermore, power consumption is quite important during the system's operation time, and thus must be considered as equally important to performance and reliability. In other words, the design goal conflicts would be more complex with more conflicting goals.

Chapter 3

Problem Definition

After the technology turn by the power wall, there have been myriad hardware/software designs to tackle domain specific issues focusing on one of the power efficiency, performance, and fair sharing. Because of the significance of power efficiency, many designs were suggested with notable changes over previous approaches. Sometimes, the changes triggered some design goals conflicts unexpectedly.

This kinds of unexpected design goal conflicts could be handled if a design could take policies for conflicting goals without significant modifications on mechanisms. The necessary condition would be good separation between mechanisms and policies for a specific design for a design goal. It could be achieved much easier with explicit resource indexing/addressing.

In the first section of this chapter, we will discuss how this design strategy can solve conflicting design issues as we discussed in Chapter 2. Then, we will discuss design issues when we apply explicit resource indexing on the last level shared cache design to resolve conflicting design goals.

3.1 Logical addressing mechanism to combine the multiple design goals

3.1.1 System design requirements and combining different goal driven designs

Even though a system can sometimes be designed for a special purpose, there must be many detailed requirements. We call them system design goals in this work (also, aforementioned special purpose is one of the goals). The system design goals are power efficiency, performance efficiency, fair resource sharing, programmability, etc. Interestingly, as we will discuss below with details, other design goals could fail to meet the requirements while focusing on a specific goal.

Case study: leakage power control and fair resource sharing

Let's extend some of discussions we made in Chapter 2. We discussed conflicting design goals between the power control mechanisms for the leakage current [41, 72, 57, 15] and the fair sharing on the shared cache [94, 84, 86, 106]. If both mechanisms were to added, the implementation overhead from cache line states and extra circuitry would be significant. The mechanisms are per a set, and increased number of states for the sleepiness status will tax the controller design significantly due to the increased state transitions on top of the states and transitions from the already complex cache coherence protocol. Inter-operation between the two is another issue because both mechanisms/policies work on the same structure.

Even though we select a power control mechanism from ones that utilizes stack algorithm, extra complexity is still left. The example of a power control mechanism with stack algorithm used on the shared cache design can be found in [15]. The stack algorithm is well established

as part of the Dynamic Non Uniform Cache Access (DNUCA) architecture. Because cache lines in the DNUCA cache are migrating up and down between cores, the distance between a core and a bank behaves the role of the stack distance. Actually, this characteristic is very good considering the implementation overhead of the stack algorithm.

However, the benefits which may work for the cache partitioning stop here. On the multi-core architecture, a cache line may need to move into additional direction when there are many hops between the cores. Then, there are multiple possible positions at the same stack distance. The multiple possible positions mean that there can be multiple search routes. Therefore, the cache line migration route and the banks for a partition must be kept. The state information for the partitioning will increase rapidly, and even more with 3D stacked architectures. Another challenge of integrating cache partitioning comes from the stack insertion point. Some partitioning scheme such as the PIPP [106] makes use of varying insertion points in the stack. It takes constant time to insert into varying positions with the flat structure in the set associative cache, but would be much tricker with the DNUCA because the exact insertion point must be traversed instead of being inserted directly.

Case study: high scaled architectures and performance driven mechanisms

The power wall issue actually changed the trajectory of frequency scaling, so performance cannot be easily boosted from the frequency scaling. Rather, chip designers are supposed to search for the chances of performance boost from the scaled transistor counts. Accordingly, the performance oriented designs at the cost of extra logics may not be easily scaled.

The DNUCA cache design [58] is a good example which transforms extra silicon into better performance. Better average access time on the cache memory is achieved via cache line migration mechanisms, more complex on-chip network over Uniform Cache Access (UCA) designs, etc. However, if we scale this design further and integrate it into more complex

design such as into multi-/many-cores, unexpected problems are unleashed. One of the most noticeable issues is the possible ping-pong effect. The performance benefit of the DNUCA cache stems from the migration of a cache block to the closer bank. The migration happens dynamically when hits. However, in the multi-core design, another direction of migration is added, and if hits from the different cores, a cache line can move horizontally (let's assume that vertical direction is farther and closer from a bank). More complication comes from the search path for a cache block.

Naturally, a method to limit the degree of freedom for a cache block movement is required with an accompanying search scheme. Possibly, a cache partitioning can be helpful to limit the degree of freedom of cache line migration. The idea is explored in [51]. To keep the dynamic placement of a cache line, the degree of freedom in partitioning is restricted. Because of separate partitions, the design is in need of global lookup mechanism which functions as a cache of address tags located at the equally farthest hop from each core.

Because the implementation cost is notable, we may consider different strategies. In [20], the authors suggested the use of high speed transmission line to access the equally farthest located cache structure to search for a tag. The idea of high speed transmission line is increasing wire dimension to improve characteristics for data transmission. So, the cost is extra area for the transmission lines and driving circuitry. Another direction is giving up the cache line migration totally. Reactive NUCA cache design [47] interacts with OS to figure out which ones are private pages, and which ones are not. In case of state changes between being shared and being private, the related pages must be shoot-down and then being re-mapped. Private ones will be put to the local cache while shared one is interleaved throughout sharers. Considering the initial accolade of the DNUCA designs for the impressive performance, it is really unfortunate to see that the increased complexity might limit the usage of the original DNUCA design ideas.

Before closing this short survey, I will list other expected issues with high scaled architectures/chips. Pin scaling issue [92] was shortly discussed in sub-section 2.2.1. The key observation behind the pin scaling issue is the cost. External pins and interconnection to them are not scaling well, but most of the extra pins from scaling must be invested to the power pins. As a result, data bandwidth would fail to catch up the performance scaling. In other words, performance would be limited by the pin scaling.

Programmability will be another critical issue with high scaled architecture, especially when a designer selects various heterogeneous computing components to achieve power/performance efficiency. For example, as widely adapted, GPGPU can be integrated on chip. The GPGPU is originally developed for processing high definition 3D graphic. Pixels can be calculated independently, so thousands of threads can work independently to handle high definition 3D image in realtime. The benefits of computing power on fine-grained data parallelism in the 3D graphic image is brought into generic programming language such as C with API called CUDA[80]. Because of the heterogeneity from the generic CPU architecture, it has distinctive structure of coding style such as stream programming, SPMD style programming, etc. Even though the APIs can be used in the generic programming, a programmer must run how to program the GPGPU.

3.1.2 Merging different goal driven mechanisms and explicit resource addressing

If integrating different mechanisms are very difficult, then we should rethink our approach to the system design. One way is to make a system extremely goal driven. However, except embedded applications, generally a system design has various specifications, so a designer tends to search for designs with specific goals to satisfy the specifications. For example, a HPC is designed for performance basically, but strong power efficiency is another important

goal of a design because we need to achieve the goal of the limited power budget, 20 MW by 2018 for a exascale system while 120 MW of power budget is expected considering current trajectory of technology trend [44, 87]. Without considering expected power efficiency issue, design scaling is a challenging issue because we might not simply scale performance with a proven design.

Let's think about a design scaling example with a specific goal driven mechanism to figure out a proper approach to handle the challenges. Surprisingly, we can find one easily. A skyscraper design is a good example of scaled design over multi-story building. There are many significant differences between the two. Due to the high scaled floor counts, the capacity increases significantly over a multi-story building. As a result, the methodology to handle traffic during rush hour differs a lot. Design decisions related to elevators are one of the important decisions to handle the high volume of the traffic. For example, we may want to add a high speed elevator considering the increased floor count. However, a high speed elevator would charge more cost to install than a regular one, so we can decide to install how many of each only after a balance between speed and throughput is explored.

It would be very inefficient if a high speed elevator stops at every floor. We should take the characteristics of the traffic into account to set up stops for a high speed elevator. The same applies to the regular ones with different degree of skipping floors to improve efficiency. We can raise a question of how the characteristics of the traffic are made. A skyscraper can be divided into multiple sections for a business, shopping, and living area where many floors are held. Within a business area, a company can take multiple continuous floors for the efficiency of co-work. From the observations such as the total traffic and traffic pattern from each section, we can choose stops for a high speed elevator. A stop at each section, or sometimes at a big business units can be placed. A traffic characteristic also can be utilized to handle the traffic in and out of the building. Of course, the roads around a skyscraper must be adjusted for the increased traffic after construction. However, adding more lanes

are not easy, so we may want to manage the bursting traffic to improve utilization of roads nearby by adjusting commuting time between sections or big business units.

A performance efficient mechanism (a high speed elevator) is integrated well in the skyscraper. Assigning stops and adjusting commuting time slots are the result of policies which can be changed over time. The reason behind the successful integration of a high speed elevator can be found from the flexibility in management. More specifically, the flexibility is observed in two directions. One is from the flexibility of adjusting stops and commuting time, and the other is multiple policies on the same mechanism. Actually, adjusting commuting time is part of resolving bursting traffic issue of in and out-of the skyscraper, but the part of policies for it is implemented without conflicts on the elevator traffic control. Therefore, we can point out the reason for the successful design integration in the skyscraper is due to the flexibility. Then, where comes the flexibility? The flexibility comes from the fact that we can control the mechanism (setting stops for the high speed elevator) with "explicit" address to index resources (here, the floor number).

The examples of explicit resource addressing actually can be found between the architecture designs. Architecture designs with Dynamic Voltage and Frequency Scaling (DVFS) [68, 45, 46] are some of them. In the architectures, voltage and frequency are scaled on demand. For example, when more parallelism is required, one can run as many concurrent threads as possible and adjust frequency and voltage to satisfy power budget. When performance boost is required for a specific workload or thread, then the selected ones get more power while others get limited power depend on the overall power budget. All the managements are possible because the core ID and thread/process ID are known already.

However, unfortunately, the idea of explicit resource addressing is not universal in the architecture design. For example, a cache memory system, especially, the one with set associative cache memory is generally hidden from the programmer's view. So a cache controller manages cache blocks reactively, in other words, when being accessed. For the respective, a

request on a cache line should be searched in a set of cache blocks because the cache blocks are kept from the access history, not from the programmer's intension. The nature of the cache memory system - the requested cache line must be searched for - actually makes it difficult to integrate other designs for specific design goals without significant design change. From now on, the focus of discussion will stay on the shared cache memory design to explain the benefits and design challenges of the explicit resource addressing to integrate with other goal driven designs and how to scale one.

3.2 Design tradeoffs

Applying explicit resource addressing on the shared cache memory design can bring many benefits. We will explore how the design challenges we discussed in Chapter 2 can be handled by adding explicit resource addressing. Then, our discussion will reach out the design issue of adding explicit resource addressing on the shared cache memory.

3.2.1 How extra resource indexing can handle design challenges on the last level shared cache

In this sub section, we will examine the benefits in some cases we discussed previously. How multiple goal driven mechanisms can be handled with explicit resource addressing comes first. As the name indicates, the explicit resource addressing works similarly to the virtual addressing which is designed to give abstraction for the main memory. So does explicit resource addressing on the shared cache for the shared cache. Therefore, similar benefits can be expected on the scaled shared cache in the handling of traffic.

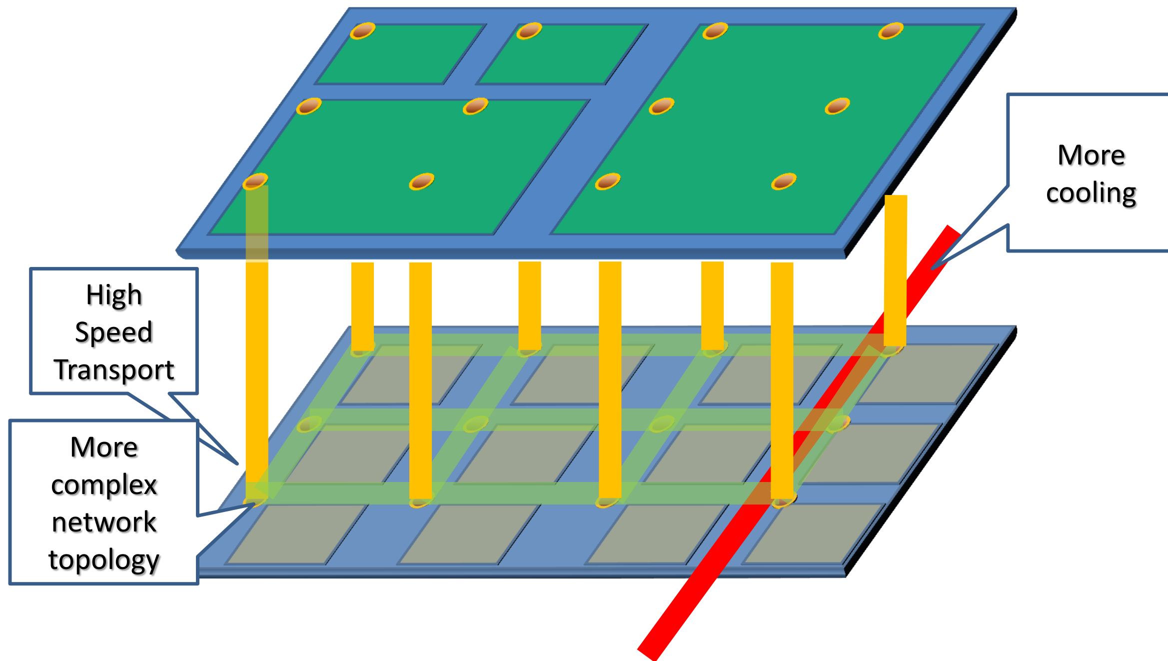

Figure 3.1: Example of future chip architecture.

Explicit resource indexing and integrating multiple goal driven mechanisms

In the previous section, the difficulty of merging the two mechanisms of power control and cache partitioning was discussed. In the closing of the discussion, cache partitioning mechanism itself can be challenging when we bring the idea into the multi-banked cache configuration. Let's see how explicit resource indexing can be helpful for the design challenges.

Before start, we'd better define the target architecture for the discussion. The target architecture is a chip scaled enough to consider all the challenging issues of power efficiency management, fair resource sharing, etc. The idea of 3D stacked architectures/chips are becoming more and more popular as we can see the examples in [105, 48]. Therefore, a chip

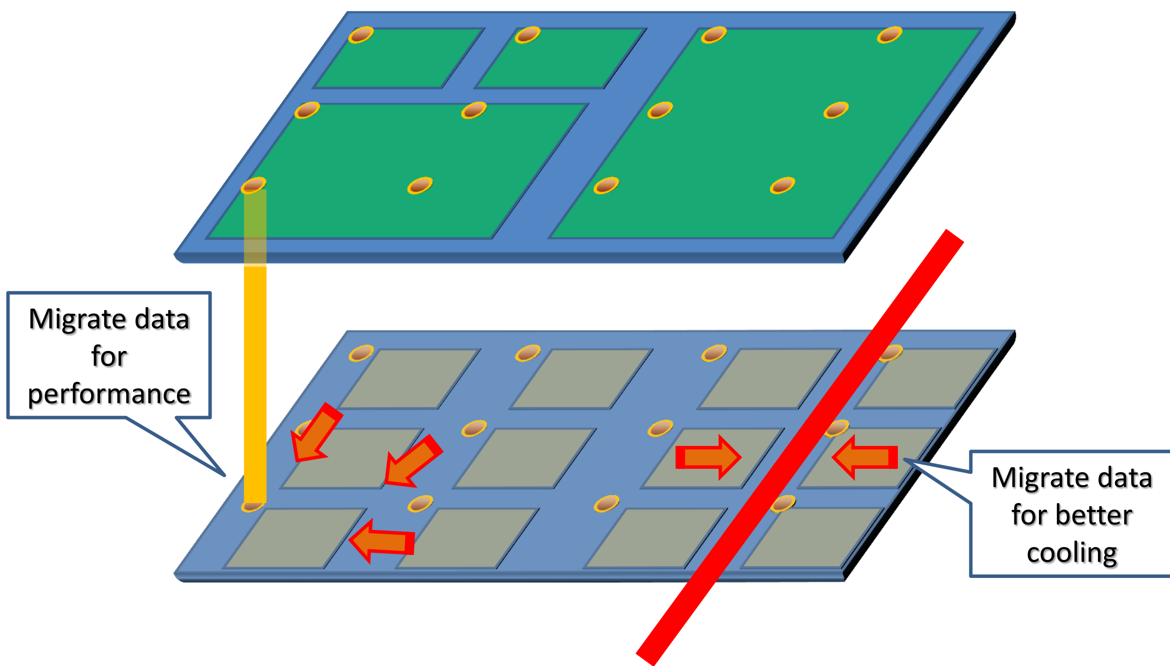

Figure 3.2: Examples of benefits with the extra resource indexing is used.

with multi-stacked and heterogeneous architectures can be a good candidate. Figure 3.1 depicts an example with many structures when scaling is considered. Different sized green colored boxes stand for heterogeneous computing components such as multi-/many-core architecture, GPGPU, etc. High speed transport such as Through Silicon Via (TSV) between layers would be helpful to handle performance sensitive traffic between layers. I also assume the use of heat pipes for cooling efficiency on the multi-stacked chip. In the case of the cache memory, multiple banked cache designs interconnected with the 3D mesh network are considered. Because of the TSV, links are heterogenous: vertical links of TSV are faster than the links on the same layer.

The possible benefits on the raised issues are depicted in Figure 3.2. We can relocate performance sensitive data nearer to the high speed transport such as TSV for the better performance efficiency. This relocations would result in smaller residency of the performance sensitive traffic on slower links. On the while, by moving frequently accessed data nearer to the heat pipes, we can achieve better cooling efficiency. Two types of mechanisms with different goals can be integrated without significant design change due to any of them. Of course, there would be interference between the two mechanisms. However, the virtual addressing - resource indexing, here index to each bank - provides medium to implement mechanisms which can adjust the interference.

The fair resource management can be added while dynamic data placement is still possible. Figure 3.3 shows an example of cache partitioning. A partitioning policy works on with explicit resource addressing via mapping each partition onto a set of banks in the example. Again, because of the virtual addressing, we can still implement data relocation in order to improve performance together with cache partitioning. This can be achieved by limiting the domain of possible mapping by sharing the partitioning information when applying data relocation. Similar strategy can be applied for power saving. After limiting placement of data to make banks which are not accessed or rarely accessed for the time being, we can

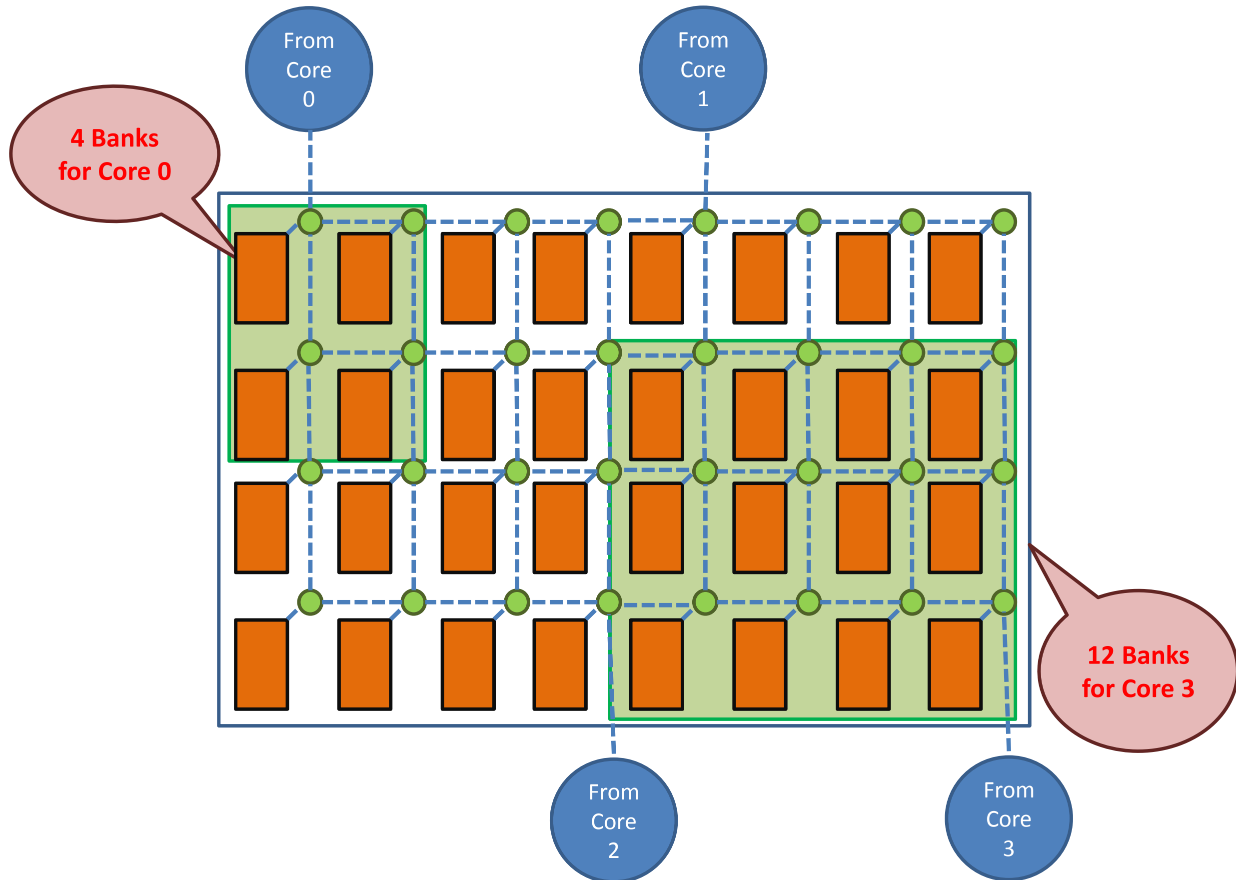

Figure 3.3: Example of cache partitioning.

turn off them for better power efficiency. Again, the power control mechanism can share the partitioning information to avoid unfair resource stealing by turning off banks on a partition blindly.

Explicit resource indexing and traffic control

The next design domain to apply explicit resource indexing is where traffic control is required. The benefits are quite similar to those of the example of the skyscrapers. One big difference is the abstraction. We human beings have intelligence to adopt the changes from relocation of the offices, or the commuting route to take from the changed stops of high speed elevators.

So, we do not have any problems with static addressing only if a well updated directory can be found at the front of a skyscraper. However, data, especially cached ones are different. Each has its tag, but the tag indicates only where it is originated from. Explicit resource indexing works similarly to virtual addressing which provides abstraction in order to separate its placement from the location in the cache memory. This kind of abstraction lets a cache memory controller relocate data.

Another chances for the benefit can be found with the issues from the limited pin scaling discussed in the previous section. One of the solutions discussed is utilizing data compression [9]. This scheme can be translated as a method to increase capacity of a link without increasing pin counts or operation frequency. A method from explicit resource indexing can find chances from different direction. Now that we can index internal of the cache memory, a block transfer or a method similar to Direct Memory Access (DMA) [42] can be used, and this will improve a link utilization between the chip and the main memory. We might enjoy the amount of benefits more when a high speed transport could be applied just like the case of the high speed elevator in the skyscraper.

Explicit resource indexing also gives opportunities to improve performance on the issues related to internal traffic. One good example can be found in the reactive NUCA [47]. The key idea of the cache design is to map private data onto private partition while shared data onto sharers' partitions interleaved. By doing so, we can avoid bank conflicts when a shared chunk of data is mapped on a single bank.

Another benefit on the internal traffic control is more freedom in the choice of on-chip network topology. Without explicit resource indexing, the possible design option is the interleaved cache design. A part of address field (generally least significant bits next to the internal block addressing bits) is selected as an index to a bank. Especially, when the bank counts are small, this approach is quite efficient especially with a simple ring topology. A ring topology has some good characteristics of fixed round trip latency, efficiency on broadcasting

by piggy-backing, etc. So, this design approach have been utilized to tweak the performance of cache coherence [16, 66], and even popular choices for the commercial chips such as the Intel Sandy Bridge chip [45]. However, with a design shown in Figure 3.1, scaled hop counts makes the use of simple network topology difficult. Additionally, lots of heterogeneity with link latencies, types of computing devices, etc. would require careful handling of traffic which would be very difficult without explicit resource addressing to manage the cache memory.

3.2.2 Design issues when integrating extra resource addressing on the last level shared cache

Adding explicit resource addressing scheme on the shared cache memory has its own implementation cost. As we can see the example of virtual memory, adding abstraction to the placement of data requires significant design efforts. Also, there can be wide spectrum of design choices depending on how much amount of design will be implemented with hardware/software. Therefore, the virtual memory design took some time to be adapted into the domain of personal computing in '80s. We remember Bill Gates said that 640 KB ought to be enough for anybody [22]. Interestingly, some Intel chips on the market have 128 MB of Level 4 cache memory [5] which is bigger than the average size of the main memory in late '90s when Microsoft's multi-processing OS, Microsoft Windows prevailed personal computing market. Therefore, with capacity scaling, our view on the efficient use of cache capacity could be changed, so with addition of explicit resource addressing.

Like the virtual memory, applying the explicit resource addressing on the last level shared cache is exposed to the similar issues. Internal fragmentation issue of paging is similar to false sharing of a cache block. To keep the mapping information, careful caching strategy is required to reduce overhead of accessing a table to keep large mapping information. Those issues actually come from the issue of granularity to manage mappings.

Design issue of explicit resource indexing: granularity of a mapping

Mapping granularity is sensitive to the performance and capacitance efficiency. If we enlarge the granularity of mapping, some chances of performance improvement comes from increased utilization of a communication link and spacial locality. However, the chances of fetching data which would not be used increased, so we generally keep smaller granularity for the smaller sized storage for better capacity utilization, and bigger granularity for the bigger sized storage expecting better performance. Now, we can have bigger sized shared cache, we may have a different granularity to manage the last level shared cache.

Here is an example of benefits from increased granularity of data placement. In [105], the authors increased management granularity to a size of a page (4 KB) between the last level shared cache and the on-chip 3D stacked main memory. Increased granularity is in order to utilize high speed transport (more specifically, to utilize the TSV), and access efficiency on the DRAM memory. Still, the shared cache itself is managed with a cache block granularity of 64 Byte, so significant design efforts were spared to perform operations with a page granularity on the cache memory with a block granularity.

With respect to the idea of explicit resource indexing, there are some cache partitioning designs on the multi-banked shared cache memory [56, 21] which mapped each cache blocks onto each bank. Both methods made partitions on the shared cache by assigning some banks onto a process. A bank is actually very big chunk of data to be managed at once, so each scheme takes its own strategy to manage overhead. Bank-aware Dynamic Cache Partitioning [56] sub-partitioned a bank into a granularity of a cache way. Considering many chips on the market, the total cache ways - the degree of a set associativity generally 16 or 24. So, the approach actually manages the last level shared cache with 1/16 or 1/24 of a bank. Jigsaw [21] is a scaled-up version of a uniform access cache memory with cache partitioning. A bank can be shared between different partitions but the internal partitioning of a bank is managed

by a local controller dynamically. In case of adjustment on the bank mapping, moving a big chunk of data might be required. Page mapping is also managed for this purpose.

As we discussed, choice of proper granularity is tightly coupled with goal specific designs. All the examples here tend to prefer bigger granularity. Another important design issue with the explicit resource indexing is related to the Operating System. If our design choice of granularity hits the size of a page, it is the same granularity used in the virtual memory system. Of course, if a designer intentionally utilizes some features from the virtual memory system, the granularity can be a useful interface to communicate with the virtual memory system. For example, Jigsaw [21], the authors made use of the page mapping in order to detect shared pages. This is possible because a page miss/fault event is generated when a page is accessed newly from other process or core, and desired functionality was added at the miss/fault handling mechanism. In the following, we will discuss a design issue when an explicit resource addressing on the shared cache memory is inter-operate with the OS.

Explicit resource indexing and the virtual memory system

In this part of discussion, we will take a look at performance issues of the virtual memory system focusing on the its caching mechanism, Translation Look-aside Buffer (TLB). Then, design issues related to the explicit resource indexing on the shared cache design will be presented.

As the essence part of the virtual memory system depends on the logical to physical address translation, the performance overhead must be compensated by the efficient caching of the address pairs. For the purpose of efficiency caching, most of CPUs on the market have a TLB to keep temporal copies of the virtual to physical address mapping pairs which are read from the table in the main memory which is called the page table. So, the performance

efficiency of this caching system would come from the rate of TLB misses over instructions or TLB accesses.

TLB misses are very effectively decreased with increased reach of a TLB. To keep the table size small, larger granularity (larger sized pages) are very helpful to reduce the miss rate [33]. However, with some exceptions, 4 KB pages worked efficiently with tested workloads in '90s. The sizes of workloads have increased dramatically over a decade, so there have been many approaches to increase the reach of TLB caching. TLB cache adopted multiple layered design to increase TLB reach while getting marginal average access time increase. Bigger sized pages such as 2 MB and 1 GB sized pages are added as recent Intel Chips [53]. Suggested approaches are not enough for some server workloads as discussed in [17]. The authors showed that 51% of execution time was spent with 4 KB pages while 10% of execution time was still taken with 2 MB pages in their experiments. Therefore, to reduce the counts of TLB misses even more, a hybrid scheme mixed with segmentation was suggested and tested.

Other challenges come from the recent popularity of parallel system designs such as many-/multi-cores. Because a single page table entry can be cached at multiple TLBs, a synchronization issue called TLB shoot down [4] is raised. A TLB shoot down is required when there are any changes on the page table. Stale mappings must be removed from every TLB, so they must be shoot down by an OS service routine. Because OS services are generally interrupt driven, a TLB shoot down also depends on the same. However, the performance overhead of inter-core interrupt is really expensive, so the authors of [102] measured that 24% of total execution time was spent only on the TLB shoot down event in their experiment. There have been many works which suggested solution for this issue. Many of them utilized a design of shared last level TLB to build a synchronization operation without the use of expensive inter-core interrupt. The followings references utilized shared TLB structure, but

did extra contributions. The impact of shared TLB entry misses and its predictable misses are discussed in [23]. Performance impact on the false TLB shoot down is discussed in [102].

By the way, discussed designs can make inter-operation difficult between the virtual memory system and the explicit resource addressing mechanism on the shared cache memory. If an extra resource indexing manages mappings with 4 KB pages, larger page sizes in the virtual memory system can give extra complexity to manage the difference. Even further, page miss event itself would not exist in some Operating System designs [32] or will be very rare [17]. New hardware mechanisms such as shared TLB cache designs also would give design issues of how to interface with the explicit resource addressing.

Actually, designs such as the OS with 64 bit single address space [32] and hybrid approach with segmentation [17] are quite different from TLB based solutions [23, 102] in the design directions. The former considered TLB caching as a performance overhead while the latter tried improvement on the TLB. If we want our explicit resource addressing mechanisms to interface with the virtual memory system, we should assume one of the design directions.

Chapter 4

Data Shepherding Cache Description

Data Shepherding cache design is my solution to add an explicit resource addressing mechanism to the last level shared cache. The Data Shepherding Cache design assumes multi banked cache structure, and a page is a basic unit of management in each bank. A page is where each cache line will be shepherded. We can add various policies on the Data Shepherding Cache in order to achieve a certain purpose, such as, power optimization, resource sharing management, performance optimization, etc.

If most accesses are within smaller hop counts, better average access time can be achieved, which means better performance. If most accesses are within a set of cache banks, we can control power consumption of other banks by power gating at the unused banks, which can be translated into better power efficiency. To implement mechanisms of the listed functionalities, the Data Shepherding will provide a good abstraction for the various mechanisms to share and to be built on.

In the following, we will describe the details of data shepherding cache design.

4.1 Overall Data Shepherding cache Design

In this chapter, we will suggest solutions to the key design issues raised in the previous chapter. Especially, the key features of the possible solution must have a strategy for good data mapping and its logical addressing with respect to banks. As wire delays become a more significant design issue with new feature sizes, more hop counts and more banks for the bigger sized caches would be more popular choice than longer wires and smaller bank counts. So, the degree of freedom of data placement is important because the degree of non-uniformness in delays to different banks grows as the count of banks increased. With this new design trend, traditional placement in a set cannot easily achieve the expected placement freedom. If stacking technology becomes more popular, then it is possible to see a whole layer of shared cache which consists of many banks interconnected via mesh network. In this cache configuration, fault tolerance and power control mechanisms will seem to be added per bank instead of smaller granularity such as cache line considering the overhead of logics to be added.

The traditional placement - placement freedom in a set - actually has very nice modern twist for the future large multi-bank shared cache. They are cache designs such as NUCA caches [51, 15, 56, 47, 77], which enhance performance by migration of more frequently used cache lines into the vicinity of cores. We can see this approach as the degree of freedom extended in a cache set, but the actual placement is abstracted away, so we do search for a line when special handling such as power control is required. The search time is proportional to the number of hop counts multiplied by the tag access time, so the applicability to the strongly scaled cache design such as stacked cache design is in question. The size of cache line itself is another issue with the traditional placement. In case of adding power saving mechanism such as power gating, we may want to add it per bank to reduce the amount of logic and complexity. However, it is not easy to decide which bank to turn off because cache lines in a bank can be shared by many cores or threads. To get a sense of enough confidence to turn

off a bank, we need to inspect line by line, or add special mechanisms only to get a level of confidence.

We can conclude that the granularity and the freedom of mapping are some of design keys for the future cache memory design. The followings are main design features of Data Shepherding cache. The first two bullets are about granularity and the freedom of mapping. The other bullets are features for the design efficiency.

- **Increased mapping granularity.** A page is selected instead of a cache line to reduce overhead of storage to keep details of mapping.
- **Mapping to a bank.** This is not mere extension from a set conceptually. The degree of freedom can adapt network topology and cooling demand(with the information whose location is closer to heat pipe, etc.).
- **Utilizing existing architecture design.** Many CPU designs have a structure called Translation Look-aside Buffer (TLB) [42]) which is a mapping table between virtual address and physical address in a page granularity. I added a field on each entry of the TLB to keep the bank mapping information.
- **Utilizing scratch pad memory.** Another design decision is the use of scratch pad memory for each bank. With bank mapping information, the location of a cache line is known already. So, it is required to track whether a line is cached (read) or not.

To implement above ideas, we need various mechanisms with different functionalities. Let's think about the design impact of larger granularity first. Because a cache line is a useful unit considering the size of data types consumed in CPU and the network bandwidth (and its impact on turn-around time), I decide to limit the role of bigger granularity. A page is bound to a specific bank, while transactions in the cache memory system are still handled

in a unit of a cache line. So, in the shared last level cache of the Data Shepherding cache, each cache line will be shepherded to a specific bank where a page is mapped.

The followings are mechanisms related to mapping between a page and a bank.

- **Data structure for mapping information details.** The expected size of mapping table is not negligible, but also the issue of locating data in the cache accurately is a significant design challenge because bank mapping information is shared and available to all the CPU cores. To reduce this overhead, I decided to utilize a TLB mechanism, especially a multi-level TLB.

- **Placement/replacement of a page mapping.** Whenever we need a new mapping, a victim must be selected and replaced with a new mapping. However, because of larger granularity, the write-back process is more complicated.

- **Routing mechanism for a cache line transaction.** We can consider the page mapping information as a part of address used only in the cache memory system. Based on the bit field of a bank number, on-chip router will forward a transaction to a correct destination.

A bank mapping between a page and a bank is done only when a new page is brought into the TLB, and the mapping is gone when the TLB line is evicted from the TLB cache. Even though most of mechanisms are hardware only, but the state of a bank mapping is kept in the TLB, the mapping information can be visible instead of being hidden in the case of the set associative cache design. In this sense, I would like to describe this explicit address for a bank mapping as logical addressing. Because the bank mappings are visible now, combined with extra information related to power control mechanism or usage information, we can implement other mechanisms for the purpose of power saving, resolving unfair usage of shared cache between cores, etc. without notable interference from each other. The figure 4.1 describes the overall data placement mechanism with the Data Shepherding.

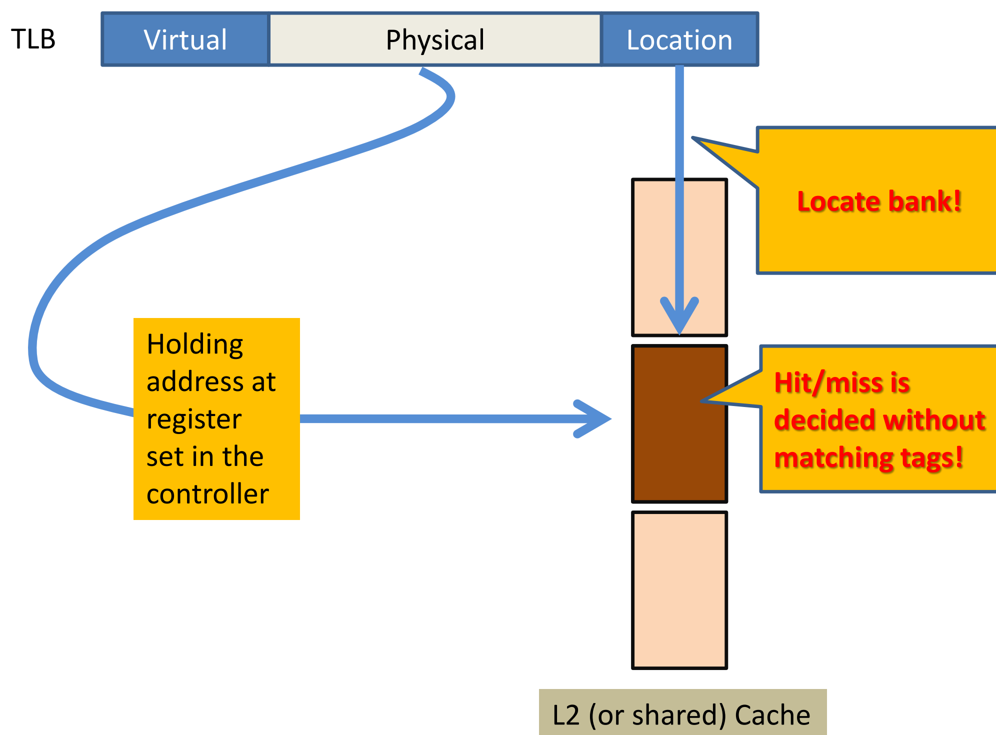

Figure 4.1: cache line placement of the Data Shepherding Cache.

Additionally, a cache line location in the last level shared cache is known during the virtual to physical address translation, so we can simplify (or ditch out) comparator/multiplexer logics to find a matching tag in a set as we can see in the set associative caches. Therefore, we can utilize a scratch pad memory for each bank instead of set associative cache. A scratch pad memory is more area efficient; accordingly access time on a bank will be reduced, and better power efficient can be achieved. We will discuss the benefits with more details about this in Section 4.4.

In the following sub chapters, we will discuss each mechanism with more details.

4.2 Detailed mechanisms for Placement and Replacement

The efficient design of the internal resource addressing mechanism is quite similar to that of a skyscraper. For each employer to go to the office where s/he is hired, correct address is known to s/he, and also, s/he may need to refer to a building map. A new office may move in, or one of the offices may want to move out. In any case, the in and out must be managed and recorded by the build management department. In this section, we will discuss mechanisms to keep the management of the bank mappings, and how to find the bank mapping information.

4.2.1 Retrieving bank mapping information and TLB cache design

Keeping a bank mapping as a field at the TLB entry will save efforts to keep the page address because both a bank mapping and a physical to logical address look-up have a page address as a search key. So, during both the instruction fetch and load/store stages,

a bank number also can be retrieved together with physical address from the TLB cache memory during the process of virtual to physical address translation. To accommodate a bank number, I expect additional hardware overhead from the increased bus width. However, the incensement is marginal compared to the size of address field. We expect bank number would not scale beyond hundreds due to the network latency might reduce the performance benefit, so the incensement would be marginal considering the address field size/word size of 64 bit computing. Therefore, implementing Data Shepherding mechanisms on the TLB cache is necessary in order to avoid redundant design efforts of having duplicate tag addresses as key values.

Bank address look-up will be done during the virtual to physical address translation, so we have a mechanism to correctly find the location of a cache line. One thing to note here is that this does not mean that we can find data at the shared last level cache all the time. The Data Shepherding mechanism only guarantees the location of a page in the shared cache where a cache line refers to. Just like the fact that knowing location of an employee does not mean that the employee is in the office now.

Two kinds of implementations are possible with this issue. One is keeping the concept of "cache miss." Let's keep a valid bit for each cache line, and set the value to 0 when not cached, but set to 1 when data is read from the main memory. By making a look-up to this bit, the shared cache memory controller can decide whether to send a request to the deeper memory layer - here, the main memory - or not. Another possible direction is utilizing a bulk or block data transfer between the shared cache and the main memory. Consecutive access on the same page performs way better in each memory module (of the main memory), so the second approach might be a good way of improving performance. However, I decided to depend on the first one because I want to measure the baseline performance of the Data Shepherding cache memory. The goal and limitation of my evaluation methods are described with more details in Chapter 5.

Before we move on to the next topic, let's further discuss the tag array of each bank. Even though the Data Shepherding cache utilizes scratchpad memory for each bank, but it is impossible to remove the tag structure entirely and easily. As we discussed, a valid bit is required the tracking whether a cache line has cached from the main memory. A dirty bit is needed to check whether the copy has been updated or not. Also, cache coherence status must be kept. As discussed in [91], many more states than the base states are needed to describe all the transient states, but the transient states are generally kept at the Miss Handling Registers (MSHR), the states at the tag entry can remain fewer in number. MESI cache coherence protocol has 4 base states, so let's assume the required amount of bits at each tag entry is 2 bits. At least 4 bits are required as a tag for each cache line in the Data Shepherding cache.

How about a set associative cache? Many Intel chips support 32 GB as the maximum main memory size, so more than 29 bits are needed for the tag address field. Also, we need various status bits as discussed, and on top of that, a bit or several to keep MRU (Most recently used) information in order to find the least used cache entry when a replacement is needed. As suggested in [96], I selected 42 bits for the Static NUCA cache simulation. Therefore, storage requirement for the tag array is reduced to 4/24 of that of a set associative cache memory, and also we do not need a complex multiplexer and comparator design as in a set associative cache, which lets us significantly reduce the size and complexity of a tag array. This will be further discussed in Chapter 6 with experimental results.

I will now discuss how to bookkeep a bank number for each page.

4.2.2 Management of a mapping between a page and a bank

Generally, a page table is managed by the Operating System, and each TLB entry will cache an entry from the page table. However, in the Data Shepherding cache, the mapping between

a bank of a shared cache and a physical page address is additionally required, and we want to use it to locate a cache line in the shared cache. For this reason, it is good to hide the bank mapping management details from the view of the Operating System while there are design choices whether the bank mapping is visible to the Operating System or not. This fact leads to some design implications. First, a bank mapping must be removed or updated when a TLB entry is being replaced, so a bank mapping is shorter lived than a virtual to physical address mapping. Second, a bank mapping must be available immediately to every core on the CPU. In the case of a virtual to physical address mapping, one is read from the page table and cached at a TLB entry in case of TLB miss. So, the synchronization point is at the page table. However, in the case of a bank mapping, it must be known to other cores if a page table entry is cached to other cores' TLBs. In other words, the Data Shepherding cache may require dedicated synchronization mechanism.

These two design implications can be realized together if we make a coherent TLB cache design. Let's assume that there are individual TLBs (a single level local TLB cache, or a multi-level local TLB cache) and a shared last level TLB cache. Each local TLB is coherent via shared last level TLB. Synchronization mechanisms to perform TLB shoot-down, etc. would be built on it. Then, we can define the entry point of a bank mapping and the exit point of a bank mapping. When a TLB miss occurs at a local TLB, the miss handling mechanism looks into the shared TLB. If another miss happens, the page table walk mechanism starts to work to read from the page table in the main memory while the Data Shepherding cache main controller manages bank mappings. If there is no room for the page table entry which is read from the page table, an entry in the shared TLB cache must be selected and replaced with the new one. So, the exit point of a bank mapping is when a TLB entry in the shared TLB is being replaced.

As we discussed in Section 3.2.2, various coherent TLB cache designs [102, 23] have been suggested as a solution to the performance issues of TLB synchronization managed by the

Operating System. Coherent TLB is a solution that reduces the overhead of TLB synchronization. There are other strategies that reduce the frequency in [17, 53]. As we discussed, the choice of coherent shared TLB cache is made because it is more directly applicable to the bank management - the life cycle of a bank mapping.

A bank mapping - more specifically, mapping between a page and a bank - raises an issue. Modern processors such as the Intel Sandy bridge [53] have bigger sized TLB entries such as 2 MB and 1 GB than the 4 KB size page. This design is intended to reduce the overhead of expensive page table walk because each page table walk actually accesses the page table in the main memory. Bigger granularity pages are helpful for reducing the frequency of page table walks, and thus can reduce the performance overhead. In the Data Shepherding cache, each page must be mapped to a specific bank, which means that all the data from the page which is to be replaced must be flushed to the main memory at the end of a bank mapping life cycle.

Flushing a big data chunk back to the main memory would be a significant performance overhead. It will also be very inefficient if big blocks are only partially used because expensive shared cache (compared to the main memory in the view point of performance and size) will be left under utilized, which may lead to poor performance. One solution is to have small set of banks that are mapped to a super page with 4 KB mapping used within the small set of banks.

Still, the write-back overhead is bigger than that of the set associative cache. In the case of a 64 Byte sized cache line, a 4 KB page requires write-back of 64 entries of cache lines. Fortunately, there are multiple optimizations. First, we can reduce the performance overhead by a block transfer between the last level shared cache and the main memory. Each DRAM bank is optimized to the page size, and so the consecutive memory accesses have much smaller access latency. Second, if a page is not “dirty”, then we can silently drop the victim page. Actually, there is a dirty flag in a TLB entry, so we can track whether a page has been

written. Obviously, the real performance impact of a page flush is important for properly evaluating the performance of the Data Shepherding cache. To model a page flush accurately, I will provide a more detailed discussion in Section 4.3.

The last thing we will discuss in this subsection is replacement policy. Now that the placement of each page is visible, various mechanisms from various design goals can be added without specialization on the cache design from each goal specific mechanism. For example, by limiting the placement of pages within specific banks, we can achieve resource sharing control. For better power efficiency, we can control the placement of pages and thereby select which bank can be turned off. Sometimes, a user or Operating System will want to control these mechanisms in order to achieve goals of performance, power efficiency, resource sharing, etc. Therefore, the degree of programmability by a user or Operating System is an important design issue. However, the issue of programmability is actually beyond the scope of this work because it focuses on exploration of the characteristics of Data Shepherding cache.

The replacement policy additionally requires information to choose a victim. For example, in a set associative cache, MRU (Most Recently Used) information per cache line is used to decide which cache line is the least recently used (LRU). Because there are 64 cache lines in a 4 KB page, at least 64 bits per page can be used to keep the MRU information if we make the hardware overhead the same. A size of 64 bits is large enough to record access time without any compromises. We can consider this storage space complexity in a different direction.

As all the mappings are managed by the shared TLB, the actual size of shared TLB entries in the Data Shepherding cache system is the same with the size of the last level shared cache divided by the size of a page. For 8 MB shared cache, we need 8 MB / 4 KB = 2048 entries in the TLB. For comparison, Intel Haswell chip [53] has 1024 entries of the shared TLB for a 4 KB page size. By the way, the Intel Haswell chip in the reference has 3 layers in the cache

system, and the last level shared cache (L3 cache) size is 8 MB. Although it may seem that the number of TLB entries increases faster than that of other chips with set associative cache architecture. However, set associative cache requires extra storage to keep the information for the replacement which the Data Shepherding cache does not need. Therefore, we can assume that the storage overhead has moved to the shared TLB from the shared last level cache in the Data Shepherding. In other words, the space complexity is on par with the set associative cache.

With the Data Shepherding cache mechanisms in this section, the last level shared cache design becomes simpler, and can run much cooler. More discussion about the last level shared cache configuration will be provided in Section 4.4.

4.2.3 Description of Replacement

The key role of the Data Shepherding controller is the replacement operation. To make room for a new page entry, a victim between a group of pages is selected while any forward progress of dependent instructions is stalled, and then all the data of the victim page is flushed out to main memory. The following is the list of operations required for a replacement in a sequential order.

Selecting a victim. If there is no room for a new page, then a victim must be selected to make room. This decision is made from the Data Shepherding controller, located at the shared TLB ,when the shared TLB controller fails to find a requested virtual to physical mapping.

Blocking. During replacement, dependent instructions and transactions on the victim and the new pages must be blocked from further progress. This procedure includes synchronization of the blocking between the cores.

Initiating a page flush. Each local Data Shepherding controller starts a page flush once one is requested from the main Data Shepherding controller at the shared TLB.

Handling a page flush with the granularity of a cache line. 64 requests (if 64 Byte for a cache line, and 4 KB for a page) of flush requests are made towards the cache memory system.

Unblocking. Once a page flush operation is done, then all the stalled actions are resumed.

A victim page is selected from a group of pages that functions similarly to a way in a set associative cache. A bank number is not a part of the request address field. Each bank of the Data Shepherding cache is a scratch pad memory, so a page is directly indexed in a bank once being mapped in a bank. Therefore, we can consider the number of banks similar to the degree of set associativity. As discussed from the previous subsection, bank mapping can be done with different replacement policies from different goals, so the size of a group from which a victim is selected can vary. If strict limitation of resource sharing is enforced, the victim candidates for a victim are limited to the pages from the selected banks.

Once a victim is selected, some synchronization actions are required. Coherent TLB is selected for the Data Shepherding cache design, so we can track which local TLB has accessed the victim page. In the Data Shepherding cache, even though a TLB entry of the victim page has already been evicted from the local TLB, the shared TLB cache may want to keep the local TLB as a sharer so that the Data Shepherding cache controller can issue a page flush request later in the event of page flush. The reason for this is because there can be cache lines from the victim page even though it has already been evicted from the local TLB.

Of course, there is cure to avoid issuing page flush on the shared core whose TLB entry is already evicted with extra hardware mechanisms. When a TLB entry is evicted from a local TLB, we can assume that there has been no cache lines touched on the page of the evicted TLB entry. Therefore, by setting a mechanism which observes the load/store queue for a

duration of time in order to ensure that no related requests are not in the queue, a coherent message could be sent to the central Data Shepherding controller at the shared TLB cache to indicate the core as not a sharer of the page any more. We can consider this method as active TLB coherence update. Of course, this method is not enough to fully declare as not being a sharer because there can be some cache lines in the local cache, extra logics are needed to track whether the shared data is in the local cache. This method would improve power efficiency of page flush, so is left for the future work.

Whether we make TLB coherence update for the future update active or passive, it is important to properly handle the data from the victim page in the local cache. To flush out all the shared cache lines from the victim page, we do actually search for the cache lines in any places of local storage/cache. A 4 KB page consists of 64 cache line sized data chunks (if a cache line is 64 Byte), so 64 searches and flushes at the local cache are required. If we know any cache lines from the victim page are not cached, then we can save power of 64 local cache accesses, and if we are lucky, we can immediately start to flush out a page from the bank in the shared last level cache to the main memory. To decide whether the victim is cached to the local cache memory, we can utilize a mechanism used at [75]. The mechanism of interest is counters for the region access. For each region being a key and a correspondent counter being a value, the authors designed a hash table to track the count of accesses. Once the count reaches zero, and then we can conclude that no cache lines from the region are cached locally. In case of the Data Shepherding cache, we can consider region as a granularity of a page. The size of a hash table can be decided empirically and fall on the area of performance optimization. So, we will discuss again in the chapter 8 as I will leave it as a future work.

Once a coherent action of a page flush is requested, a flag is set at the victim TLB entry so that progress of any instruction fetches or data accesses to memory system can be stalled or blocked. Actually, this blocking procedure is the same with page table walk. The difference

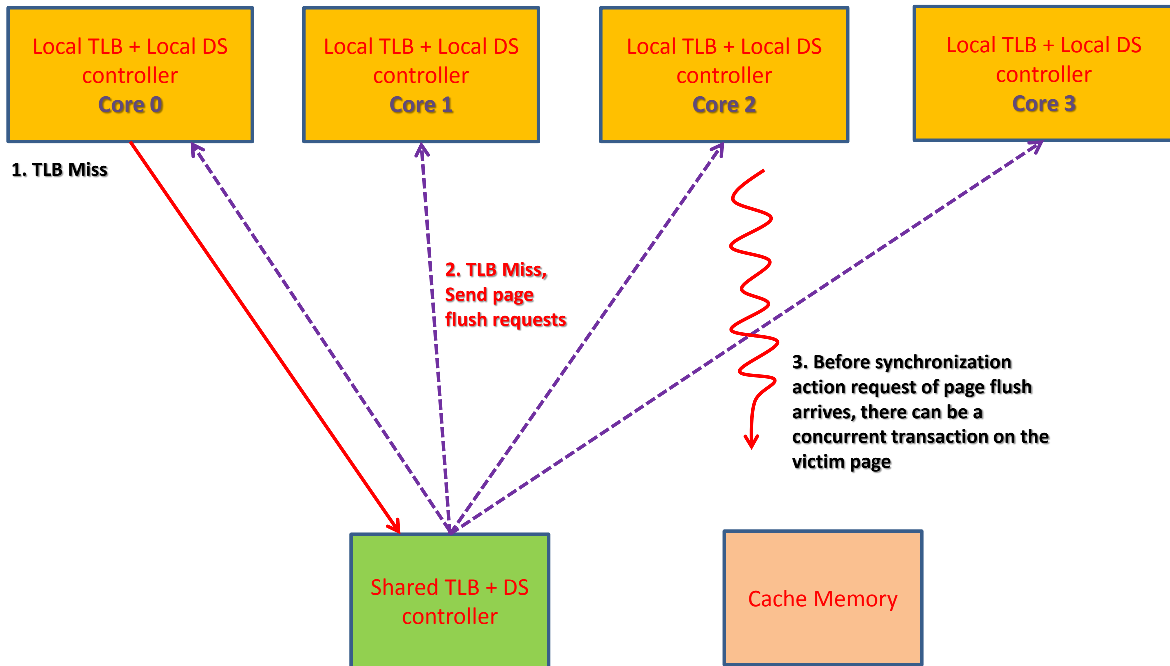

Figure 4.2: Race between coherent page replacement and requests to memory system.

is that now the operation is triggered as a part of page flush process. The cores which have accessed the victim page will do the same blocking and flush operations.

However, a subtlety comes from the difference with page table walk: the blocking action at the Data Shepherding cache is built on a coherent shared TLB design. As depicted in the figure 4.2, a race can happen. Even though the possibility is low, but it can happen because the requests already issued to the load/store queue can proceed together with a page flush. The detailed scenario of this race is as follows:

A TLB miss from a local TLB is forwarded to the shared TLB. At this point, we don't know which page will become a victim page. Of course, because of bigger granularity, the chances for accessing on the least recently used page is very low.

A victim is selected for a replacement. Then, flush requests on the victim page will be sent. As discussed above, there can be some cache lines locally cached from the victim page. So, flush requests must be sent to every core which has history of accessing the victim page. There can be requests which are under service when a victim is selected. Also, new read/write requests can start between the TLB miss and the arrival of the page flush request.

A coherent page flush request arrives. Requests from now on can be blocked on the victim page.

The design decision I made here is letting the concurrent requests complete. In other words, overlap the execution of a page flush with the execution of already issued requests on the victim page. As discussed above, if a core shares (have been accessed the victim page) the victim, to flush out all the data from the victim, we are to inject 64 requests of cache line flushes (for 4 KB page granularity, and 64 Byte sized cache line). When a coherent page flush request arrives, local Data Shepherding cache controller caches it to a special set of registers called Page Flush Status Table (PFT). Each entry of PFT (PFTE) tracks the progress of a page flush. Each PFTE has a field for the victim page address as a key, and 64 bits to track 64 cache line flushes as a value. If the local controller queries a cache line address in the victim page to the load/store queue, and if no requests in the load/store queue is on the queried cache line address, then a bit pointing the queried cache line in the PFTE is set and a flush request for this cache line is injected towards the cache memory system. If request(s) on the queried cache line is/are found, a query for different cache line - one can be the next cache line in order or one with value 0 at the PFTE - will be issued in the next cycle.

This design decision yields following design issues. First, as we do not expect the exact time for the request at the load/store queue being serviced, the local Data Shepherding controller may look up continuously. We can make it look up the load/store queue periodically. Other approach for this issue is to let the load/store queue notify the completion to the PFTE. Second, increasing throughput of a page flush can be expensive. To increase throughput of a page flush, in other words, performing multiple page flushes at the same time, we need more ports at the load/store queue. As we stay baseline performance of the Data Shepherding cache to figure out the characteristics of a page flush, the possible design for multiple page flushes and its evaluation will be left as a future work.

4.3 Page flush mechanism

We will start the discussion on the page flush mechanism from estimation of the performance impact. For the purpose, we will examine performance sensitive mechanisms related to the page flush. After pointing out the page flush mechanism as a major performance challenge for the Data Shepherding cache design, I will start discussion on the overall page flush mechanism with respect to the cache coherence mechanism. In the last of this section, I will describe the specific details on the each layer of cache memory system.

4.3.1 Estimation of performance impact due to page flush

Let's start discussion with the mechanisms - mechanisms for the page flush - whose performance impact are expected to significantly influence on the overall performance. By doing so, we will have better sense of relative overhead of each, and will have more understanding on the characteristic of the page flush on the performance.

1. **Coherence between TLB caches.** Whenever page miss event occurs at the shared TLB, Data Shepherding Cache (DS) controller is kicked in to select a victim to replace it with the newly requested page. In the case of replacement, the victim must be dropped from every private TLB cache. As discussed in Section 4.2, we are building Data Shepherding cache with coherent multi-level cache design, so, actual performance overhead and extra hardware efforts are mostly on the replacement policy to select a victim. However, this procedure can be overlapped and negligible if the TLB miss triggers page table walks or page fault.

2. **The Frequency of the Page Flush Event.** Recently, bigger granularity TLBs have been added to improve performance by reducing the frequency of Page Table walk. For example, Intel Sandy Bridge[17] has three different sizes (4 KB, 2 MB, and 1 GB). If DS mechanisms work only with 4 KB sized TLBs, it cannot enjoy the benefits of bigger granularity TLB entries. As we discussed in Section 4.2, by mapping a large page onto small set of smaller sized banks, we can enjoy the benefits of large pages. However, this only reduces the frequency of the page walk - the frequency of the page flush stays the same.

3. **Stalling read/write requests on the victim page.** Because the page replacement of Data Shepherding only update internal addressing, once a bank mapping is changed, then every transaction from the victim page which is selected to make a room for a new page must be flushed out or blocked from being fetched. Blocking mechanisms can share with TLB miss handling mechanism, but the transactions which are already issued to the load/store queue must be handled correctly. Suggested solution is overlapping a page flush with the transactions under service as discussed in Section 4.2.

4. **Overhead of write-back to main memory.** If a cache line size is 64 bytes and we are flushing a 4 KB sized page, then 64 flush requests are to be issued. If every cache line in a page is touched, then we can consider this as bulk replacement. However,

only a cache line is touched from a page, and the page becomes a victim, then 63 cache line flush requests are injected into cache memory system, which is a pure performance waste.

In short, the obvious performance overhead comes from the frequency of the page flush events and the overhead of write-back to main memory. They seem to a really big performance loss against the design without the page flush such as the Static NUCA cache memory (SNUCA) [51], especially when the frequency of the page flush events stays unchanged with the super page. However, the following performance estimation gives us it would not be that serious, and sometimes there would be chances the Data Shepherding cache could perform better.

If the probability of the amount of cache lines used in a page is 0.5 and it is uniformly distributed, we can estimate that 32 flush do real write-back to the main memory on average. Also, if we assume that the cache memory system is perfectly pipelined, then 32 cycles per a page flush is the expected performance overhead from a page flush. By the way, in the real world, because of bank conflict, if a bank were not sub-banked enough, perfectly pipelined cache is very difficult to implement. Therefore, the discussed inequality of equation 4.1 forms low bound of performance overhead.

$$T_{Pageflush} \times (64 - Ave_{lineflush}) < Overhead_{Total} \tag{4.1}$$

$T_{Pageflush}$ is for the total count of page flushes, and $Overhead_{Total}$ is for the total extra performance overhead from Data Shepherding cache in cycles. $Ave_{lineflush}$ is the total number of cache lines which are actually flushed out to main memory. The constant 64 is used under the condition of 4 KB page size. If $Ave_{lineflush} = 64$, which means most of the time each

assigned page is used fully, i.e. every line in the page is dirty, then there will be no extra performance overhead expected, but it would be a really extreme case.

On the other hand, there are performance gains. As we will discuss at the section 4.4, average access time of the Data Shepherding cache is expected to be smaller than that of the SNUCA cache with the same size. If we keep the assumption of perfectly pipelined (or with no bank-conflict) cache memory system, then we get following equation of overall performance estimation.

$$Perf_{DataShepherding} = Perf_{SNUCA} + Overhead_{Total} - AccessCount \times \Delta AccessTime \quad (4.2)$$

$AccessCount$ is the total count of accesses into the cache memory system, and $\Delta AccessTime$ is the difference between average access time of the SNUCA cache memory, and that of the Data Shepherding cache memory system.

We cannot directly perform the analysis using the equation 4.2 and average access time. As discussed in [49], real memory system is very complex, average access time or other memory system performance can only be decided from experiment. For example, the limit of outstanding misses, the degree of sub-banking, the number of hop counts to reach designated bank, etc. affect average access time and performance overhead and the effects are dependent on the workload and program, so only can be decided imperially. Also, actual access time varies based on the types of coherent message. Types of accesses - read/write, sharing pattern, and network topology will be influential to the average access time.

In conclusion, to evaluate the performance characteristic of the Data Shepherding cache, we need decent amount of details on the memory system, especially related to the page flush mechanism. The detailed design will be discussed in the remaining of this section. Before

leaving this sub-section, I want to point out there is another aspect of performance analysis for the Data Shepherding cache memory. Because of larger granularity of management, the last level shared cache can suffer fragmentation as we can see in the main memory. However, some of Intel Xeon chips [27] on the market already has 45 MB of L3 cache memory, and this is actually bigger than the main memory of common desktop PC in late 90's. Interestingly, late 90's is a decade after Bill Gates told 640 KB would be enough to anybody [22]. Fragmentation can be an important issue, but considering the size scaling of the last level shared cache memory, I think it is giving priority to the modeling of page flush. I leave fragmentation measurement as a future work.

4.3.2 Page flush mechanisms and cache coherence protocol

A page flush is a quite different process from read/write request into the cache memory. In case of read/write request, requested data must be fetched from the main memory if the requested one is not cached yet. On the contrary, the page flush makes one directional flow of data. From this fact, we can take design benefits in two ways.

Firstly, as discussed, we can design flush mechanism in a cascading manner from the top to the bottom layer of the cache memory system. Two actions are expected when a cache line flush has arrived.

- **When a cache miss is declared.** We can forward the cache line flush request to the next level of the cache memory system hierarchy.
- **When a cache hit is declared.** Not only forwarding the cache line flush request but also flushing out data are required. In the case of a clear cache line, we can simply drop the line and forward the flush request to the next level of the cache memory system. Of course, it depends on the details of a cache coherent protocol used. Here,

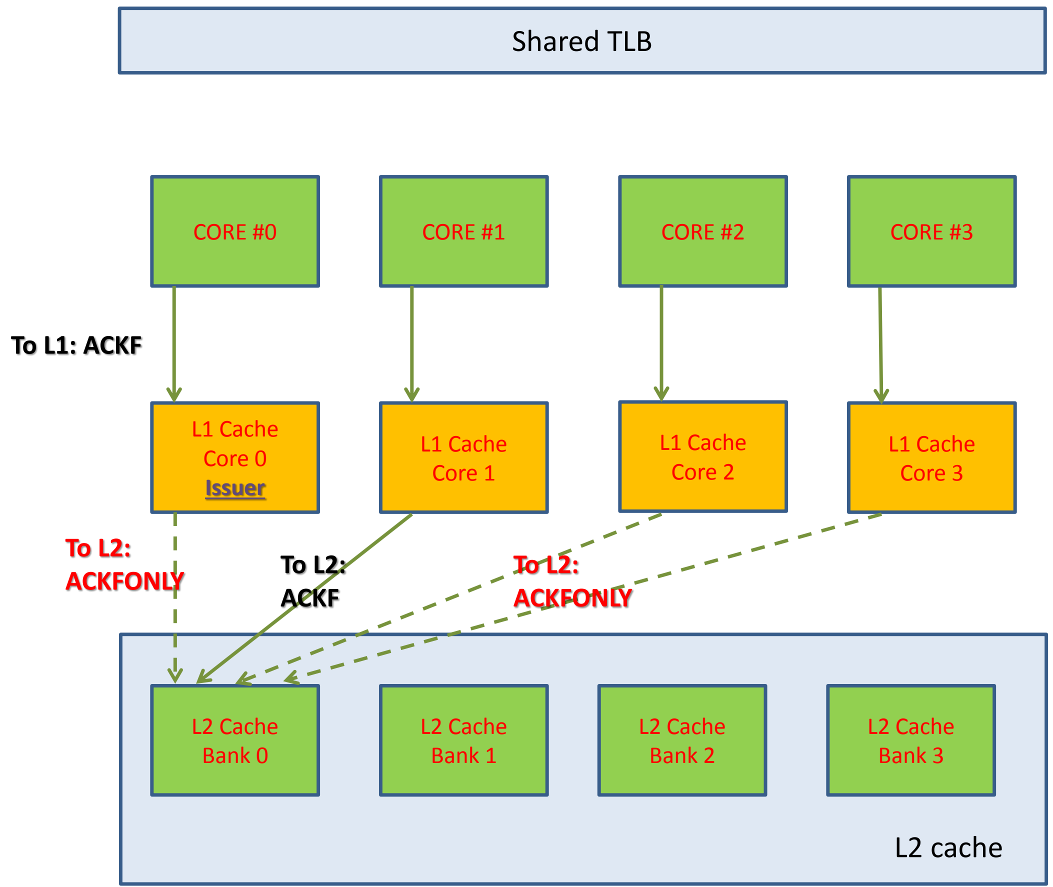

Figure 4.3: Overall flow of a cache line flush.

we can do a performance optimization. Instead of making separate requests, we can piggy-bank data with the cache line flush request to the next level due to the cascading style operation.

An example of the overall process of a cache line flush is depicted in Figure 4.3. 4 solid lines between cores and L1 caches are a line flush request, named as `ACKF`. Between L1 caches and the shared L2 cache, we need two types of requests. `ACKF` is used for the case of cache-hit and data piggy-back, while `ACKFONLY` is used when a cache-miss or a clean cache-hit occurs. Let's assume that the cache line is in the exclusive state (E) in the MESI cache coherence protocol [35]. The local cache of core #1 has the exclusive state line, so cache hit is declared, and a new message with piggy-backed data and `ACKF` are injected towards the L2 shared cache. Other 3 cases between the L1 caches and the shared L2 caches are `ACKFONLY`. This process will remove the overhead of multi-/broad-casting to the upper cache memory layer to keep coherency if a replacement process must perform in case of a set-associative cache memory system.

Secondly, we do not need to worry about blocking accesses from the flush process. As explained in [39], a cache memory can block from further accesses when it is under service. Because a cache miss leads to a round-trip to and from the next level of the memory system, in worst case, a cache memory can be locked down until data arrives from the main memory. To avoid this performance overhead, generally Miss Status Holding Registers (MSHR) are used to keep the status of request under service from the lower level memory system to release the resource hold. Actually, the MSHR are on the critical path of servicing requests, and also must be content addressable, so the size of the MSHR is an important design issue. Actually, the flushing process is not required to be cached at the MSHR, so the page flush process is not sensitive to the size of the MSHR which decide the degree of outstanding transactions under service. Also, because of the page flush, the Data Shepherding cache does not need the cache replacement at all. This will remove some transient states [91] from

the cache replacement, there are chances for optimization to make a cache coherence protocol simpler by reducing some states. The cache coherence protocol optimization will be left for the future work.

Different from the case of a cache line flush, there are some subtle issues when we start and end a page flush. One of the subtlety comes from the fact that we need to count the progress of page flush to declare the end of page flush. By counting `ACKF` and `ACKFONLY` messages per a cache line, we can declare when a cache line is over. Likewise, if we count 64 of the cache line flush complete, then it is the end of a page flush. To initialize this set of counters, correct number of page sharers must be known. Delivering the number of sharers to the L2 cache controller can be done by sending a notification of `PAGE_FLUSH` with the number of sharers. Let's call the set of counters to keep the progress of a page flush a Flush Status Buffer (FSB). The additional functionality of FSB is keeping the page address because the Data Shepherding cache does not keep address of each cache line in the tag array - just like a scratch pad memory does.

When we start a page flush, because of the large scale (many hopes in the on-chip network) we assume, it is possible to have a race between a page flush start notification, `PAGE_FLUSH` and a cache line flush request, `ACKF` or `ACKFONLY`. The race can be a problem because the end of page flush cannot be declared if a cache line flush request is not counted towards completion of a cache line flush. The solution is to make the start of cache line flushes coherent to the initialization of the FSB. An example of a page flush start is depicted in Figure 4.4.

Similarly, there can be a possible race between a cache line flush and access request from the new page address if we declare that page flush is done after the issue of the last cache line flush request of the victim page. This kind of race can be handled by comparing the address of the conflicting requests, and then just handle them in the order of arrival in the queue similarly to the case of the cache replacement. Even though the Data Shepherding cache

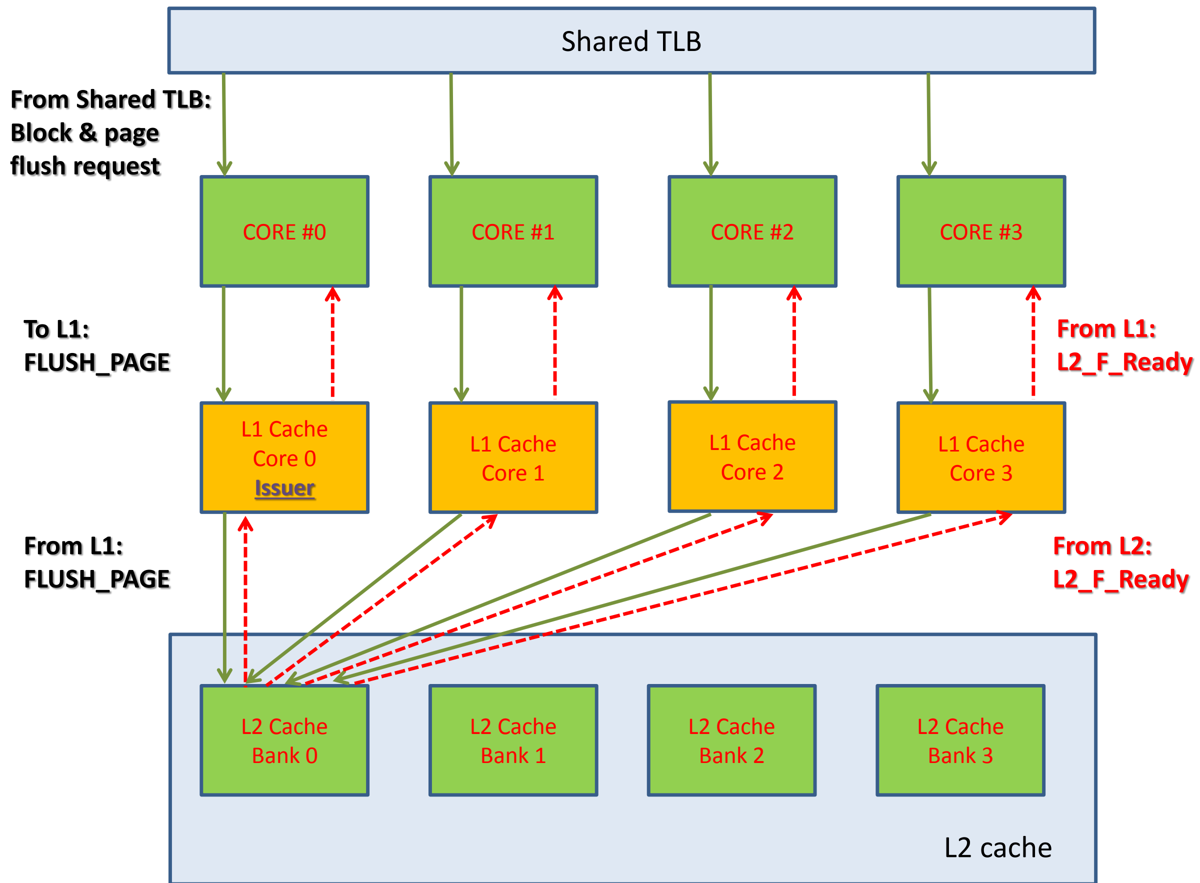

Figure 4.4: Start of a page flush at Bank 0 in the L2 cache.

does not have a tag comparison logic, but this does not mean the address of the request is not delivered. Address of the request arrives to the cache memory controller correctly, and kept while during the service in the MSHR in case of a cache miss, and at the top of the queue in case of immediate service. During the process of a page flush, the victim page address is kept at the FSB. So, by making comparison with the addresses in the described mechanisms, we can declare a page flush is done as soon as the last cache line flush request of the victim page is injected towards to the cache memory system from each core.

We assume a large scale shared last level cache with many hops, so there are chances of races including order inversion between the couple of last line flushes and requests from the new page. Of course, we can do something to recover the order, but it comes with additional complexity. To avoid extra complexities, I decided to make the completion of a page flush at the shared TLB coherent to the decision made at the bank. As depicted in the figure 4.5. Once every cache line indicates as flushed in the FSB, the controller at the Bank 0 will issue a `ACKF_PAGE` towards the upper level caches. They are forwarded upwards towards the Shared TLB. My design reveals the latency from the bank to the shared TLB visibly, in other words, without any overlapping. I think performance optimization is the next step because I want to reveal the characteristic of the cache management with bigger granularity then a cache line. So, the chances for the optimization leaves as a future work.

4.4 Physical configuration of Data Shepherding cache

4.4.1 Overview of physical configuration

A discussion about the physical layout of the last level shared cache is the focus of this section. I considered various design issues from the scaled cache size. The overall configuration is depicted in Figure 4.6. The total number of banks can be bigger. I just selected 8 just for

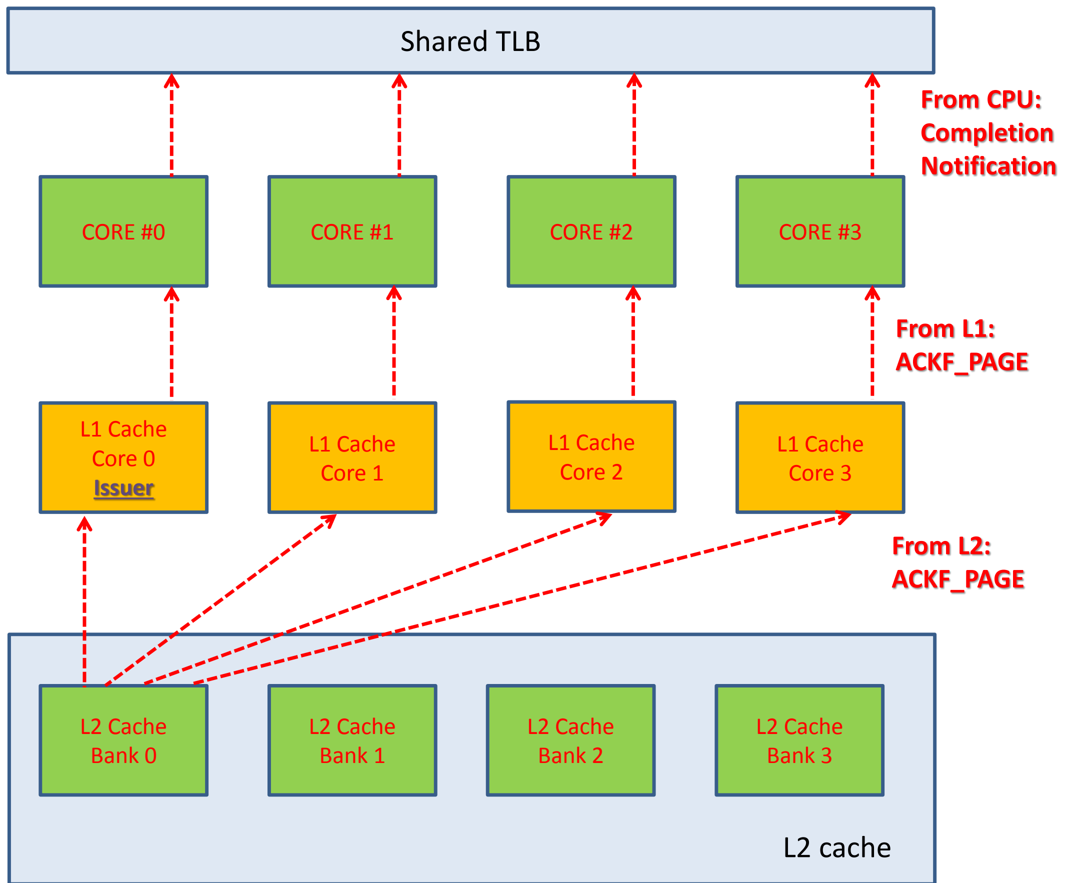

Figure 4.5: End of a page flush at Bank 0 in the L2 cache.

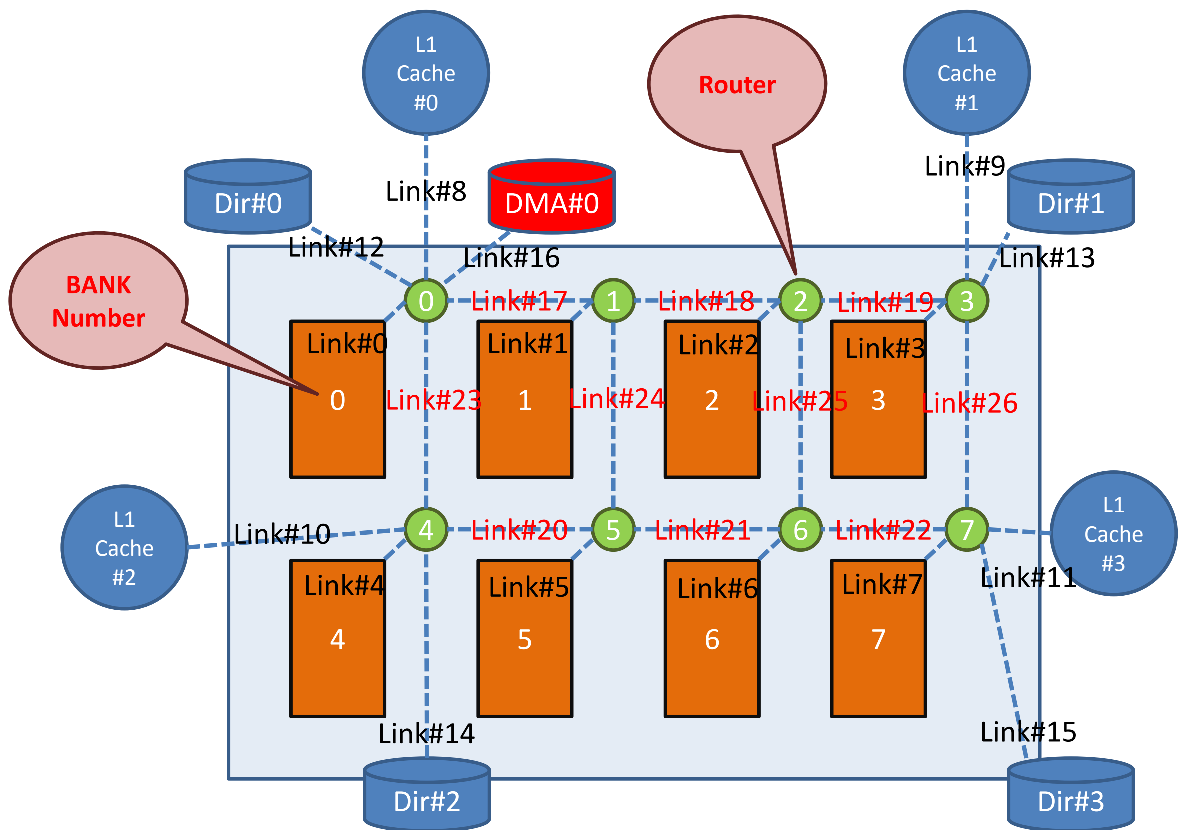

Figure 4.6: The overall configuration of the last level shared cache of the Data Shepherding cache.

drawing convenience. Because mesh topology is assumed for the large scale, Non Uniform Memory Access (NUMA) is also assumed for the main memory. 4 memory controllers handle equally separated physical address range of the main memory. Black colored links are external links to the shared last level cache, while red colored links are internal links. External links' bandwidth or link latency can be different from the internal links' considering device characteristics.

The choice of 2D mesh topology is to reflect possible future design trends such as many-/multi-core designs. Of course, the network design can be more complex if we consider stacked multi-layered on-chip memory system as we discussed the benefits in Section 3.2.1, but the benefit from the Data Shepherding cache can be applied equally. Here, I assume every internal link has the same property, but as we showed in Figure 3.1, we can have links with different physical properties. The following list of items are the implication of the Data Shepherding cache with more complex networks other than a bus or a ring topology.

1. **Increased degree of freedom on the physical placement.** A page can be mapped to one of the banks in the last level shared cache. So, the number of banks works similarly to the degree of a set-associativity. If there are enough banks, then the Data Shepherding cache can provide more degree of choices to avoid conflict misses. Therefore, we can use direct-mapped or scratch pad style memory instead of set-associative cache memory for a bank design.

2. **Early decision of dynamic routing policies.** Because the replacement decision is made at the shared TLB, we can decide a victim and a bank mapping with a policy from various dynamic demands as we discussed in Section 3.2.1. Therefore, we can get benefits of dynamic mapping, but also actual addressing into a bank can be done in a static manner by removing the possibility of searching other banks to find a requested cache line. This is a good property to keep the complexity of the router and controller

design constant over the other cache designs with dynamic mapping. The benefits were discussed in Section 3.2.1.

3. **The use of scratch pad memory managed by hardware.** The use of scratch pad memory for a bank design will not only be power efficient, but it will also keep the link latency scaling slower than that of a set-associative cache design. The reason is the simpler structure has smaller footprint, so the scratch pad memory based bank design can have shorter link length.

As the design implications of the first two items have already been discussed previously, we will focus on the details of a bank design in the following sub section.

4.4.2 Discussion on the bank design

The use of the scratchpad memory design to substitute set-associative cache memory is not new. As discussed in [13, 101, 50], the scratch pad memory has been an alternative to the cache design of embedded systems for power efficiency. Physically, it does not need extra circuitry to search for an entry in a set, so it is more area and power efficient. However, the contents of the scratch pad memory is required to be moved before the execution of related codes and must stay without automatic replacements, so the programming generally requires compiler support[100, 82] and sometimes with a high degree of hand-tuning. As a result, the design generally stays in the domain of highly optimized special purposed systems.

If a bank were to be managed fully similar to a scratch pad memory, a whole page must be transferred before its use. Considering locality, fetching a whole page is not a bad idea if critical cache line first (similar idea with critical word [49]) is implemented together. However, the benefits can be application dependent, and there can be cases wherein pages are only partially used. So, I decided to adopt half reactive approach. As we discussed,

initially each cache line in an empty page slot in the bank has a valid bit unset. Every tag field is unset and cleared while a page is being flushed. A first access to any of the cache lines in the page will experience "miss" because the status is not valid cache line, and after fetching data from the main memory, it will be set as valid. The cache lines stay as long as the page they belong to in the cache. As pointed out, block transfer (size of a page) can be efficient considering locality. Also, there is a performance benefit from the structure of DRAM modules. Generally, consecutive hits from the same page have smaller latency, so a memory controller tends to optimize performance by coalescing requests to the same page. Therefore, implementing block transfer between the shared cache and the main memory can be an interesting design issue, but I plan to leave it as a future work.

We have two arrays in a bank: one for the data, the other for the tag. Because we do not need the long decision path of tag search, we can select sequential access of the tag array and the data array, or even parallel access of both. The sequential access can achieve power efficiency at the cost of some performance. This design is popular choice of some embedded systems, as we can see in the ARM processor [61]. The parallel access suits for the modeling of a bank in the Data Shepherding cache because each cache line is directly indexed without a tag match. I selected the parallel access as the base model. Of course, the sequential access mode is expected to give a decent power efficiency, but I decided to leave it as a future work.

The last item to discuss in the modeling of a bank is sub-banking [89, 28]. Sub-banking is an approach to have multiple arrays with independent address and data lines. It allows concurrent accesses to a bank while keeping smaller implementation overhead than multi-porting. Sub-banking can be an important issue with the Data Shepherding cache design. Many of the multi-bank cache designed chips on the market such as Intel Sandy Bridge [45] utilize multi-bank to achieve interleaved accesses. On the other hand, a whole page is mapped to a single bank with the Data Shepherding cache, so workloads whose access pattern shows lots of spacial locality, then can experience more bank conflicts than a interleaved multi-bank

cache. Actually, performance loss can be compensated with shorter access time in the scratch pad memory bank, so it would be an important design issue that could be only verified with experiment. This issue will be discussed further in Chapter 6.

Now, it is time to discuss an extra design feature on the bank controller. As we discussed in Section 4.3.2, a data structure to track the progress of a page flush is required. The structure needs 64 counters (64 is the number of cache lines in a 4 KB page with 64 Byte line size), and each counter can count up to the number of cores or all the possible sharers in case of heterogeneous architecture design. We called the structure Flush Status Buffer (FSB). During the operations of a cache line flush, a counter is decreased when a message that a cache line flush from a sharer is done. Sometimes, the message will come with a piggy-backed flushed data. Once every counter reaches zero, then we can declare a page flush is finished. Because of FSB and cascaded coherence actions for a cache line flush, messages/request related to the page flush will not increase Miss Status Holding Registers (MSHR). Details are discussed in Section 4.3.2.

Chapter 5

Simulation Configuration

We have discussed the details of the Data Shepherding cache design in Chapter 4. The discussions made for the Data Shepherding cache design touch almost every layer of the memory system, so I want to limit our focus onto the accurate modeling of performance overhead. In the first section, We will discuss the selected Data Shepherding mechanisms which I implemented in the simulator, GEM5 [25] and the reasons I selected them. Also, explanations of the system to be compared will be given. In the following section, implementation details of each selected mechanism will be discussed. The final section of this chapter is about the mechanisms I added to simulate bank conflicts for better accuracy of experiment results.

5.1 Overview of experiment setup

5.1.1 Goal of simulation setup

The fundamental idea of the Data Shepherding cache is to utilize the benefits of additional resource addressing mechanism to get various benefits as we discussed in Section 3.2.1. So, a

possible way of design verification is implementation of mechanisms to achieve various goals, and make comparison with other designs. However, it is obvious that we must understand the overhead of the Data Shepherding cache separately without adding the mechanisms to achieve specific design goals. Then, we can figure out when to apply the Data Shepherding cache design or not. We will stay on the evaluation of the Data Shepherding cache design only in this work, and the evaluation with the mechanisms to achieve various performance goals are left as future works.

There are many things to consider in the design to emulate the Data Shepherding cache. I think one of the best ways to evaluate the performance characteristics is to compare performance of one with others. By the way, the performance of a component is not easy to be evaluated solely independently; we need to be very careful to model the critical path of execution so that the performance evaluation can model the performance characteristics well. Also, the Data Shepherding design is related to almost every stack of memory system hierarchy. Therefore, a proper choice of components to emulate is an important issue.

Then, what will be the cache memory system to be compared with? I think the best possible choice is one with similar configuration without strong optimization to achieve specific design goals such as power efficiency, fair resource sharing, performance efficiency, etc. I selected Static Non Uniform Cache Access (Static NUCA, or SNUCA) cache [58] to compare with the Data Shepherding cache for good reasons. Static NUCA cache design is a cache memory with multiple banks interconnected with on-chip networks such as mesh topology. Each bank is addressed statically, so the average access time is the sum of average latency to each bank and access time of a bank. Actually, Static NUCA cache design plays a role as a baseline to make a comparison with Dynamic NUCA cache designs. Also, we can consider popular ring based interleaved cache design, such as the shared last level cache of Intel Sandy Bridge chip [45], as a variant of Static NUCA.

The critical path of execution is well described in Section 4.3.1, and there, we concluded that the overhead of page flushes would be the most notable performance issue. On the other hand, because of the need for synchronization between the cached state of a page at each local TLB cache, I decided to use coherent shared TLB cache design. As discussed in Section 3.2.2, coherent shared TLB will soothe the overhead of TLB shoot-down, so extra performance gain will be added, and this will make the effect on the performance of the page flushes unclear. Therefore, I decided to assume the same TLB mechanism on both cases of the Static NUCA cache and the Data Shepherding cache. Then, if the same TLB mechanisms are assumed, we can even drop the details. So, I selected System call Emulation mode (SE mode) of GEM5 simulator for the execution of simulation. The SE mode only runs statically linked binaries because it emulates system calls without Operating System. TLB faults and misses are also handled by software (via emulated system calls), so all the detailed overhead related to the TLB are removed from the critical path of execution of the target machine and binary. Of course, the functionalities for the Data Shepherding cache on the TLB are modeled and implemented. The implementation details with the design decisions for simulation will be discussed in Section 5.2.

With the discussed change, we can still compare the characteristics between the Data Shepherding cache and the Static NUCA with respect to the effect of the page flushes, but there are limitations. First of all, the omitted details on the shared TLB design may limit the research on the mechanisms to achieve various goals. The bank mapping information can be used as an interface to implement controls from software or the Operating System, but the details of discussion for the software interface would be left as a future work. Also, the use of the SE mode actually limits the types of workloads to compare. Currently, GEM5 does not support multi-threading fully on the SE mode. So, we cannot make experiments to get the performance characteristics with multi-threaded workloads.

There are several other design decisions for simulation setups. The choice of cache coherence protocol - MESI directory CMP protocol- is selected because it is used mostly for the inclusive cache designs such as Intel processors. The version of GEM5 I used is the one with change set `9449:56610ab73040` with some patches applied by hand. There are some differences from the current version. For example, it does not have Ruby cache coherence protocols with 3 level cache. So, I worked with a 2 level cache design - the L1 cache is local to a core, and the L2 cache is the last level shared cache.

In the remaining of this section, we will discuss on the system setting of interest of the two systems: the Data Shepherding cache memory and the Static NUCA cache memory.

5.1.2 Comparison between Data Shepherding and Static NUCA cache setup

The two cache designs are different in many ways, but there are some common features. As depicted in Figure 4.6, both design share the same on-chip network (mesh topology) and the same number of banks. However, the routing methodology is different. The Data Shepherding cache maps a page to a bank dynamically, so it needs special mechanisms for the management. More particulary, for the implementation of the page flush mechanism, the Data Shepherding cache design has more coherent messages for the page flush, but at the same time, there is room for optimization because the Data Shepherding cache does not require cache replacement. The details of the added mechanisms for the page flush will be discussed in Section 5.2. Let's assume the difference of cache coherence mechanism does not increase operation time on each state transition because there will be no increase in the number of states. By the way, the number of concurrent page flushes can affect the performance of the Data Shepherding cache.

The real physical difference is made from the different bank architecture. A difference comes from the tag size. The default tag size of set associative cache in the CACTI 6.5 is 42 bits which is reflected from the trend of common physical memory size on the market [78]. In Section 4.2.1, I estimated the size of tag in the Data Shepherding cache as 4 bits. In the implementation of mesh CMP directory protocol of GEM5, there are some states that are not cached at the TBE (the name of structure in the GEM5, used for the same functionality of MSHR). From the estimation, I decided to add 3 extra bits to each. So, 45 bits for the tag size of the Static NUCA, and 7 bits for that of the Data Shepherding cache are used for the experiment setup. The details of the states are discussed in Section 5.2. Another difference in the architecture of a bank is the way of accessing each array in the bank. The Data Shepherding cache does not need to search for a match in a set, so it reads both tag and data array in parallel. Detailed simulation result from CACTI 6.5 will be discussed in Section 6.1.

If we set the capacity as the same on both designs, the Data Shepherding cache design is expected to have smaller area because it utilizes the scratch pad memory. So, access times to each bank will be different. The size difference actually can make difference in the latency of a link between on-chip routers. Considering the performance of network, we must check the parameters that are sensitive to the size of a bank such as the degree of a set associativity of the Static NUCA cache, and the degree of sub-banking on both. Of course, the parameters are related to the access latency. The experiment result will be discussed in Chapter 6.

The number of MSHR is another performance sensitive component because it decides the number of concurrent transactions under cache miss. As we discussed in Section 4.3, FSBE also tracks the status of cache line flush, so a cache line flush does not use MSHR. In GEM5 ruby memory system, `m_outstanding_requests` decides the degree of total outstanding requests to the memory system, so if we set the size of MSHR bigger than 16, then the size of MSHR at the L2 cache (the last level shared cache) will not affect the performance

because it is already maxed out. Actually, the Data Shepherding is not supposed to increase the number of MSHR because of FSBE. However, because I decided to utilize states from cache replacement (even though we do not have cache replacement at the shared L2 cache in the Data Shepherding cache), experiment setup uses MSHR to keep the flushing status of each line. Therefore, I decided to set MSHR big enough not to make any bottleneck. The default value of TBE (works as MSHR at each cache memory) is set to 256, and I kept the value for my experiment. One thing I want to put a note here is the degree of concurrency actually affect the performance. If consecutive cache line flush requests arrive at the cache, the following one must wait until the access from the first one is over if it is not sub-banked enough. I will discuss about this concurrency issue on the simulator in Section 5.3.

Lastly, the number of banks works differently on both cache configurations. On the Static NUCA, as the number of banks increases, the average access time will be increased. Of course, the Static NUCA indexes each bank interleaved way, there are some performance benefit by avoiding some bank conflicts if there are series of consecutive cache line accesses. However, the Data Shepherding cache "interleaves" pages, not cache lines, so it can more efficiently resolve conflicts to the different pages in the same set of pages, but consecutive accesses onto the same page would cause more bank conflicts than the Static NUCA cache. So, the performance of Data Shepherding cache may be more sensitive to the degree of sub-banking. The followings are the parameters which will work for variable during experiments.

Shared cache configuration: the number of banks, the shape of mesh topology, the link latency, the number of PFTE (DS only).

Bank configuration: the degree of set associativity (SNUCA only), the degree of sub-banking, tag/data array access latencies.

The followings are different bank configurations, but will be fixed during experiments.

Size of a tag entry: 45 bits for SNUCA, 7 bits for the Data Shepherding cache.

Access mode on the tag and data arrays: normal access (the tag look-up selects an entry from a set of cache lines read from the data array) for SNUCA, parallel access for the Data Shepherding cache.

The details of other system settings used in the experiments are described in Chapter 6.

5.2 Implementation details on the simulator GEM5

5.2.1 Modeling of the Data Shepherding mechanisms in the TLB

In the System call Emulation mode (SE mode) of GEM5, TLB misses and page faults are handled "magically" without performance overhead. One of the reasons of skipping the details of TLB layer is to make a fair comparison between the two system (assumes the two have the same TLB mechanisms), but another reason comes from the practical point. The modeling which depend on TLB would be huge overhead while the returning from the huge efforts are marginal to the modeling of page flushes. So, the first design issue is where to put the main Data Shepherding controller.

In the GEM5 simulator, there is a component called a sequencer. A sequencer plays a role of an interface between the processor core side and the Ruby memory system. In the GEM5, there are many models of a memory system which we can select based on the details and complexity. The Ruby system is the one with detailed cache coherence protocol. I selected a sequencer module in the Ruby memory system to put the main Data Shepherding controller features for good reasons.

First of all, every transaction into the memory system (including the cache memory system) enters into a sequencer and returns to the processing core from the sequencer. Therefore, this is a good place to add and remove addressing information to each bank for each transaction. This decision makes the simulation design simpler because there will be no further modification at the core side. Another good feature to implement the controller functionality of the Data Shepherding is that a sequencer keeps tables for read requests and write requests from the connected core, so we can track the transactions in the load/store queue indirectly. This means that we can track transactions which are already issued towards memory system on the page selected as a victim. As we discussed in the section 4.2, the already issued ones are serviced first, and then they will be flushed out eventually. The transactions which arrives after the page flush are blocked first, and they will be resumed after the page flush is done.

Lastly, because a sequencer is a C++ class, it is relatively easier to access `g_system_ptr` compared to other controllers of caches and the main memory which are written in SLICC. `g_system_ptr` points a `RubySystem` module. It is globally used between all the Ruby system components and only single entity of `RubySystem` class exits. Because we assume that the main Data Shepherding controller is at the shared TLB cache layer, it must be seen between every module of the same functionality (here, every sequencer) to model the Data Shepherding controller. A pointer is not allowed at the SLICC (of course, there is work-around by making a class to access), so a sequencer is a good place to put the Data Shepherding mechanisms at the shared TLB cache layer. By the way, SLICC is a domain specific language to generate C++ code for cache controllers used in the Ruby memory system. SLICC focuses on easier description of a cache coherence protocol.

Figure 5.1 depicts overall setup for the experiment. The functionalities to model the Data Shepherding controllers moved to the sequencer from the TLB cache layer. The biggest difference of Figure 5.1 from Figure 4.1 is a new structure, the Bank Mapping Table (BMT). The BMT is a structure to keep the bank mapping, independently from the TLB structure.

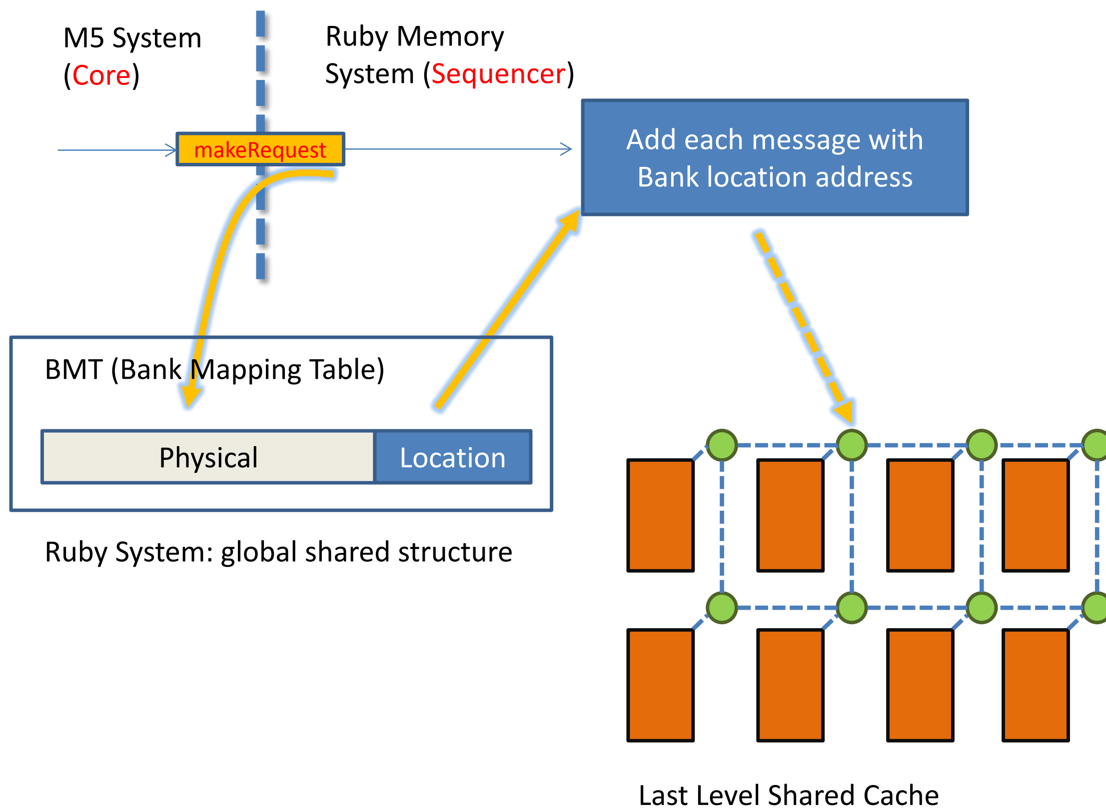

Figure 5.1: Overview of the Data Shepherding cache simulation setup

Because it is a structure for the simulation purpose, so we are open to improve simulation performance. The BMT is implemented with a hash map, an `unordered_map` in C++ standard template to improve the performance of simulation, and included as a member of the `RubySystem`.

We need other data structures for the implementation of a page replacement policy. I implemented a LRU page replacement policy for the experiment. Let's discuss when we want to implement this policy on the real hardware. Each bank in the Data Shepherding cache is a scratch pad memory, so the number of banks work as a kind of a set associativity. True LRU policy requires huge overhead especially with a set-associative cache (here, SNUCA cache). The size of an access time counter can be different with different time units, but a set associative cache such as SNUCA needs one per a cache line while the Data Shepherding cache needs one per a page. With 4 KB page and 64 Byte for a cache line are assumed, the storage requirements for the true LRU will be 1/64. Anyway, in simulations, I just selected true LRU for both to focus on the performance effect of a page flush keeping the influence of other variables minimal.

Let's take a look at the possible overhead on the hardware implementation. The Data Shepherding cache has an issue with true LRU implementation. Because the main controller is at the shared TLB, we need to update the time stamps from the local TLB. The time to collect all the update will scale with the number of banks, so it will be huge overhead. Actually, because the size of local TLB is very small because it is generally implemented with Contents Addressable Memory (CAM), the number of entries which need actual time sync will be very small. This is good news. As we observed in many LRU approximation page replacement policies such as second-chance algorithm [42], they work pretty well even though they are not using accurate time stamps. In that sense, if we make a local TLB to send time stamp update coherence action periodically, it can achieve decent LRU approximation. As I use true LRU cache replacement policy on the SNUCA cache without any hardware

overhead estimation, I used true LRU based page replacement on the Data Shepherding cache without any hardware overhead estimation.

There are other data structures for the implementation of the LRU page replacement policy. We need a storage to save the time stamp of the most recent access, and we can check all the pages from all the banks indexed to the same location in a bank. In the real implementation, all the pages indexed to the same slot in a bank will form a set in the shared TLB. However, for the performance of simulator, we have a hash table named BMT to find a bank mapping. I added extra field at each entry in the BMT to keep access time, and made a structure named In-time Counter Table (ICT) to keep the page addresses in a table with a width of the number of banks and the hight of the number of pages in a bank. When a replacement policy is needed, we can find a correct set of pages using the page address from a transaction, then find the smallest access time by iterating the BMT with the page addresses in the set. The In-time Counter Table (ICT) is originally to track the oldest page, but this features is not used for this work. The ICT is also a member of the `RubySystem`.

One thing to note before leaving the discussion on the replacement policy is about the tie breaker. In case of multiple candidates for a victim, I select the one closest (in hop counts) to the core. More precisely, I mean a sequencer in the simulation which triggers a page replacement process.

Locating a bank for each transaction

One of the big difference between the SNUCA and the Data Shepherding (DS) is how each cache design locates a bank. Because the SNUCA cache in the GEM5 accesses banks with interleaved access pattern, the next higher bits to the block size bits are used to locate a correct bank. More precisely, if a cache line size is 64 Byte, then the least significant $\log_2 64 = 6$ bits are the block size bits which will index inside a cache line, and the next

least significant $\log_2 Number_{totalbanks}$ bits are used to locate a bank. As a result, a set index starts from the next bit to the bank index bits.

We can get a bank index from the TLB (the BMT in the simulator) in the DS cache, so an index to locate a page in a bank starts right after the block size bits.

In the simulation setup, the bank index information is not carried with a request transaction. Instead, it is referred whenever addressing from L1 cache to L2 cache is required. This decision is also for the coding efficiency. In the real hardware implementation, the bank index is carried into the next layer of the memory system until the last level shared cache. By the way, even with 128 banks, only 7 bits are increased to the address field of a request transaction. Considering the physical memory of modern systems and width of a bus, I think 7 bits increase is just marginal, so we decided to exclude from the simulation detail.

Implementation details in the sequencer

Because of the nature of a sequencer which interfaces between the two system of the core (M5) and the memory system (Ruby memory system), there are some functionalities which are overlapped between load/store queues at the core side and L1 cache memory controller. Two tables named as `m_writeRequestTable` and `m_readRequestTable` track all the outstanding read/write requests from the core just like load/store queues do.

The `m_outstanding_requests` sets limits to the maximum number of outstanding transactions into the L1 cache memory. A request which can be successfully issued to the L1 cache memory will be put into the mandatory queue. A request in the mandatory queue will be picked up one by one in First In First Out (FIFO) order at the L1 cache. Successfully read data from the memory system will be returned by L1 cache controller via `readCallback` and `writeCallback` calls, and eventually requested data are returned to the core. The overall flow of transactions from and to a sequencer is depicted in Figure 5.2.

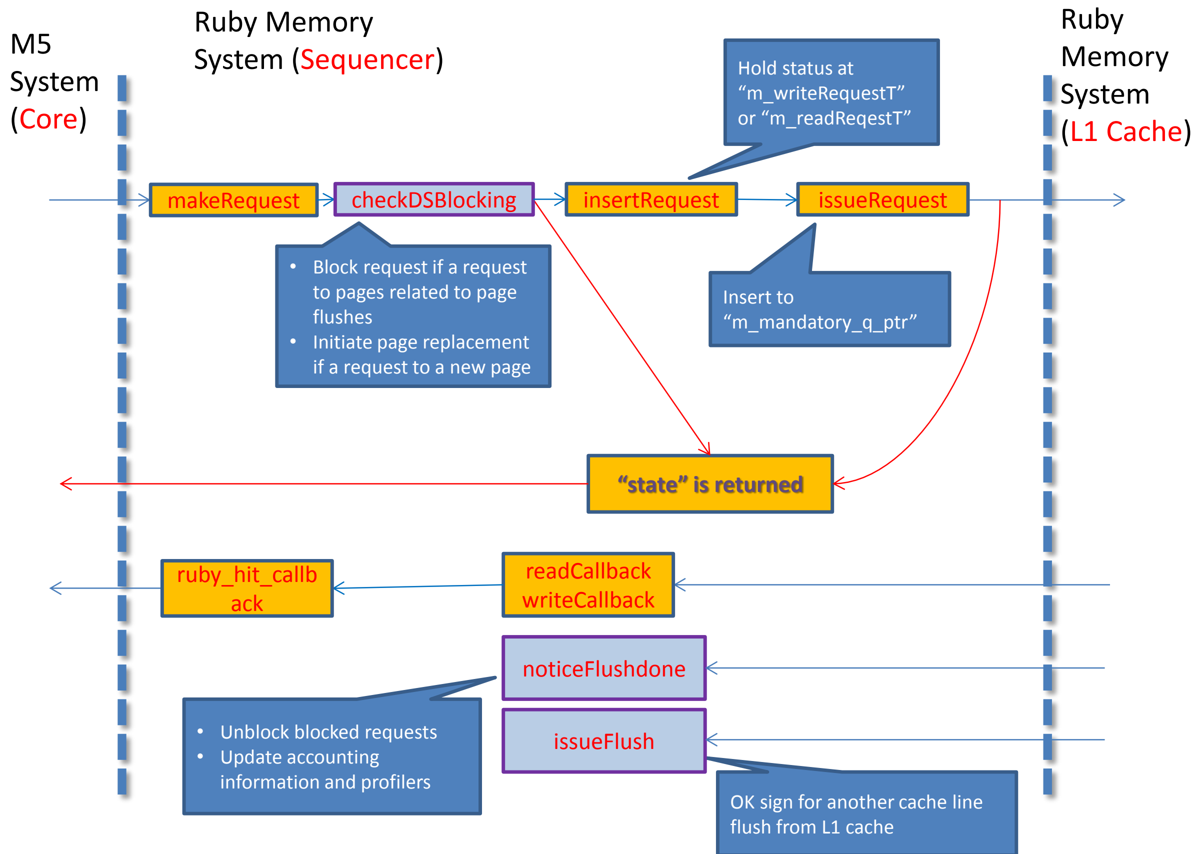

Figure 5.2: Flow of transactions from and to a sequencer.

In the figure, the orange colored boxes are the original functions, and the light blue colored ones are added to implement functionalities of the Data Shepherding controllers. Solid blue colored arrows depicts the flow of data while the red colored arrows show the return of control back to the callers.

The `checkDSBlocking` does important roles. The basic role is to check whether a request is to be blocked or not based on the status of a page in the BMT. Two types of pages can be set as being blocked. One type is victim pages. They are under being flushed, so if a request falls on one of the victim pages, then it must be blocked. The second type is pages which trigger page flushes to make rooms for themselves. Here, the first type needs careful handling. We are keeping the total size of the BMT entries to the same with the total counts of pages in the shared L2 cache. Any replacement makes extra entry for the new page. Eventually, only the new page is kept and the victim will be deleted, so the total size will be kept. To clean up victim pages, because the BMT is a hash table, we need a structure to track pages related to the page flush. I added a new structure named Ownership Counters (OC) which is indexed by the core and keeps the total number of concurrent page flushes at a core and the list of victim pages and new pages which trigger page flushes.

After referring to the BMT, we can declare a request is to be blocked or not. The blocked transactions must be kept in the separate table other than `m_writeRequestTable` and `m_readRequestTable` because the blocked ones will be reinserted to the tables when the related page's slot is established in a L2 bank (as a process of "unblocking"). The unblocking of the blocked requests will be done at the `noticeFlushdone`. The `noticeFlushdone` is the last function in the processing of a page flush. The function is called from the L1 cache controller when it receives a notification of completion of a page flush. It does other functionalities such as updating profilers and accounting information such as the number of concurrent page flush, updating the victim page list, etc.

There is another case for the blocking at the `checkDSBlocking`. If a request is under a page miss, in other words, does not belong to any of pages in the TLB cached pages or the list of pages for the blocking, then we must trigger a page replacement to make a room for a new page in the shared cache. However, a new page replacement event can be prevented in the cases which will be described. As discussed in Section 4.3, processing a page flush consumes resources such as an entry in the Page Flush status Table (PFT), the size of concurrent page flushes are set by hardware capacity. Also, we have special set of counters called Flush Status Buffer (FSB) to keep progress of a page flush to declare completion at the L2 cache. The number of outstanding page flushes for each core is counted at the OC, while the number of outstanding page flushes for each bank is counted at the Bank Access Counter table (BAC). When there are no room for a new page flush when described resources are not available, then we can return the control to the caller with notification of failure so that the caller will retry later. When it is possible to start a new page flush, a request of `PAGE_FLUSH` is issued to the L1 cache as shown in Figure 4.4.

5.2.2 Modeling a page flush

The overall page flush start and page flush completion are similar to the discussion made in Section 4.3. However, there are some notable changes from the discussion. The first change is some differences in the message types. The table 5.1 describes the extra coherence messages added for the Data Shepherding cache in the simulation setup. Different from Figure 4.4 and Figure 4.5, we can see some functions instead of messages. The reason is a sequencer is a C++ object, so messages returned from the cache memory to sequencer must use member functions of the sequencer class.

The second difference comes from the fact that functionalities of the Data Shepherding control at the shared TLB is now at the sequencer. Even though the data objects for the

Communication channels	Coherent message for DS operations
Requests from Sequencer to L1	FLUSH_PAGE, ACKF, NACF
Coherent requests from L1 to L2	FLUSH_PAGE, ACKF, ACKFONLY
Coherence responses from L2 to L1	ACKF_PAGE, L2_F_READY
Response method from L1 to Sequencer	noticeFLUSHdone()
Request method from L1 to Sequencer	issueFLUSH()

Table 5.1: Extra coherence messages for the Data Shepherding simulation.

implementation is located at the Ruby system object which is singleton object shared by all the ruby memory system components, but the operations are done locally from each sequencer. Therefore, a sequencer is made to triggers a page replacement and to update related data structures in the Ruby system object. So, a sequencer which starts a page replacement is in charge of issuing a FLUSH_PAGE message, calling issueFLUSH(), and all the related actions such as updating the list of blocked pages, unblocking pages, updating profilers, etc.

The last difference is from how to trigger line flushes. In the simulation setup, 64 line flushes are required to be issued into the cache memory system one by one for a page flush. We can insert 64 line flush requests into the queue between a sequencer and its local L1 cache, but this can unfairly add delays to the transactions that are not related to page flushes. Actually, the overlapping between a page flush and transactions which are issued before the page flush from the same page prevents issuing of the line flushes on them until they are serviced fully. In order to avoid unfair delays, each line flush is designed to be injected one by one. When a line flush request is received at a L1 cache, a request to the next line flush is sent to the sequencer of the L1 cache. By doing so, the message queue need not to be filled with 64 line flushes at a time, and we can check the status of the transactions which are under service every time when a request of line flush arrives. More discussions on this will be followed.

Implementation details in the L1 cache

Because now we trigger a line flush request from a L1 cache, we need extra design changes related to this. Let's take a look at the process of triggering of a line flush with more detail. We can start actual line flush process from the L1 cache once `L2_F_READY` is received from the L2 cache via `issueFLUSH()` member function of the sequencer class. When we call `issueFLUSH()` function, it must be called with a specific cache line address. So, it is more convenient to move functionalities of the PFT into the L1 cache. Even though the actual design of the PFT entry and the FSB entry (used at the L2 cache) are quite different, but we can consider the PFT entry as a special case of the FSB entry with all the cache line flush tracking counters set to 1 from the point of view of modeling. I reused the `FSBTable` class at the L1 cache controller to track the progress of a page flush.

Once a cache line flush request `issueFLUSH()` is called from the L1 cache controller, the function call checks `m_writeRequestTable` and `m_readRequestTable` at the sequencer where the L1 cache controller is attached to see there are any transaction on the cache line address. If no other transactions are found, then it is good to start a cache line flush. So, a message with `ACKF` is queued to the attached L1 cache, and by the time it arrives to the L1 cache, actual line flush process is started. At the same time, `issueFLUSH()` is called for the next cache line flush.

When there is a transaction on the cache line address which will be flushed, a message with `NACKF` is queued to the attached L1 cache instead. On receiving the `NACKF` message, we cannot proceed actual cache line flush of the address with the `NACKF` message. Cache line flush on the "NACKF"ed cache line is skipped, and `issueFLUSH()` is called for the next cache line flush instead.

As each layer of memory system in the ruby memory system works when a transaction arrives at the queue of incoming traffic, it is easier to look into the functionalities added for the

Message	Triggered Event	Description
FLUSH_PAGE	init_FLUSH	Send FLUSH_PAGE message to the L2 cache.
ACKF	F_ProcACK, F_ProcACKonly	Trigger cache line flush. An event is selected based on the cache line status. Also, issue a next cache line flush request.
NACF	F_ProcNACK	Cannot proceed data flush with the line address. Only issue a next cache line flush request.
ACKF_PAGE	fwd_ACKFP	Forward a page flush completion notice to the sequencer.
L2_F_READY	FLUSH	Initialize FSBE. Send the first cache line flush request to the sequencer.

Table 5.2: Extra coherent messages and events on the L1 cache.

Data Shepherding cache system with "events" which are triggered from transactions. Also, an event is an important coding structure with the SLICC script. It behaves as a input to the state transition of a cache line's coherence state. With other inputs such as a dirty bit, etc., an event is selected uniquely, and performs actions to model hardware behavior, and the coherence state of a cache line makes a transition into the next state. So, a next state of a cache line's coherence state can be described from **CurrentState** $\times$ **Events**. In Table 5.2, I listed all the events triggered from the DS coherent messages.

One notable entry is with two events from the ACKF message. The selection between F_ProcACK and F_ProcACKonly is made from the cache lookup result. If there is a cache hit with the address of the ACKF message, then the F_ProcACK event is raised. Data in the cache line is sent to the L2 bank, being piggy-backed in a ACKF message. On the while, if there is a cache miss, the F_ProcACKonly event is triggered instead and a ACKFONLY message is sent to the L2 bank without any data.

The remaining cache line flush process is similar to the process of write-back from the cache replacement. Actually, it has some unnecessary state transitions to handle coherence reply

from the L2 cache which can be removed on the Data Shepherding cache. However, I decided to make a performance comparison without any performance optimization on the Data Shepherding cache, so the Data Shepherding cache shares state transitions of the write-back process without modification.

Implementation details in the L2 cache

Extra events on the L2 cache memory controller are listed in Table 5.3. One thing I want to note here is that I added as many events as possible to make the trace run descriptive for the debugging convenience. It does not hurt performance of simulation at all due to no increase in the number of states, so I did not try any optimization of merging events.

The most significant structure for the Data Shepherding cache is the Flush Status Buffer (FSB) table. Each entry consists of a set of counters. There are array of counters with the number of total cache lines in a page, and an additional counter to count the number of cache lines which have been flushed out to the main memory. When initialization of a FSB entry (FSBE), we need to set all the counter as 0, and also need to keep the number of page sharers which are sent with a `FLUSH_PAGE` message.

Once a FSBE is initialized, the L2 cache controller tracks the progress of a page flush from the coherence messages, such as `ACKF`, and `ACKFONLY`. When we increase a line flush counter, three types of output are raised from the FSBE controller. One is indicating more messages are needed to complete a cache line flush. Another is to indicate a cache line flush is done. This case actually is separated into two cases because it is possible that the cache line being flushed is the last one in the page flush. The two latter cases trigger a write-back of the cache line to the main memory. Additional action of sending a `ACKF_PAGE` message is required when the FSBE controller raised a signal of a completion of a page flush. In each description of Table 5.3, I described the outcome of the FSBE controller in the second sentence, and the

Message	Triggered Event	Description
FLUSH_PAGE	init_PF_L2	Prepare for a new page flush by initializing a FSBE.
ACKF	FLUSH_ACK	Data is written back. Line Flush is not done yet.
	FLUSH_ACK_wb	Stale data arrived. Line Flush is not done yet.
	FL_data	Data is written back. Line flush is done.
	FL_wb	Cache line is dirty. Line flush is done.
	FL_wb_clean	Cache line is clean. Line flush is done.
	FL_DS_only_wb	Stale data arrived. Line Flush is completed.
	FP_data	Data is written back. Page flush is done.
	FP_wb	Cache line is dirty. Page flush is done.
	FP_wb_clean	Cache line is clean. Page flush is done.
	FP_DS_only_wb	Stale data arrived. Page Flush is completed.
ACKFONLY	FLUSH_ACK_old	Line Flush is not done.
	FL_done	Cache line is dirty. Line flush is done.
	FL_done_clean	Cache line is clean. Line flush is done.
	FL_DS_only	Line flush is done.
	FP_DS_only	Page flush is done.
	FP_done	Cache line is dirty. Page flush is done.
	FP_done_clean	Cache line is clean. Page flush is done.

Table 5.3: Extra coherent messages and events on the L2 cache.

first sentence describes the presence of piggy-backed data or the status of cache line whether it is dirty or not in order to show the source of the data if a write-back is required.

5.3 Resource Conflict Modeling

All the mechanisms discussed so far focused on the modeling of the behavior of the Data Shepherding cache. However, the simulation result only with the discussed mechanisms did not work well. The average latency of page flushes was surprisingly close to 64, so the Data Shepherding cache performs better then expected.

The reason comes from no bank conflict in the L2 cache bank while series of coherent line flush messages are arrived at a specific L2 bank for a page flush. There are some structures and hooks to model a bank conflict in the stock Ruby memory system of the GEM5 simulator, but the implementation on the MESI protocol (`MESI_CMP_directory` protocol in the Ruby memory system) are not matured enough to be used for my purpose directly.

In this section, we will discuss the details of a bank conflict model I implemented for the experiment. Let's call my bank conflict model the Resource Conflict Modeling (RCM). The RCM is used only for the L2 cache (as a shared last level cache in the experiment). The bank conflicts in the L2 banks are crucial to the modeling of page flush with better accuracy. The RCM on the L1 cache is not needed because we assumed that the L1 cache have small latency (2 cycles) and fully pipelined.

5.3.1 Overall the Resource Conflict Modeling design

The Ruby memory system already have some codes to model bank conflicts. There is a `BankedArray` class to keep the usage of each sub-bank so that we can query whether there

are any bank conflicts. The query is made via the `tryAccess()` method of the `BankedArray` class, and the method is wrapped with the `checkResourceAvailable()` method so that it can be used in the SLICC code.

However, the `checkResourceAvailable()` method is not used at all at the L2 cache description, so a proper model must be designed first for both the SNUCA and the Data Shepherding cache. Another notable issue is with the `tryAccess()` method. A `BankedArray` object is set in use when accessed. This may be correct in the view point of behavior modeling, but in the view point of correctness of simulation, this is not. For example, when we model parallel access of the tag array and the data array, calling for the `tryAccess()` for both `BankedArray` objects of tag and data arrays, and there are cases that one is accessible while the other is not. For better accuracy, we will use multiple access mode and multiple access time(they will be discussed later in this section). To decide correct access time, we need to inspect current status of cache line. Also, different access modes may need different kinds of access pattern and resource use. Therefore, if we want to make a decision on the access time in a single fuction of `tryAccess()`, this function will become very complex to maintain.

So, I prepared separated the functionality of the `tryAccess()` method into `tryAccessDS()`, `setAccessTimeDS()`. The `tryAccessDS()` method checks whether access is possible. The `setAccessTimeDS()` method is to set resource usage of a `BankedArray` object. Also, a new method, `getResourceAvailableTime()` is added to insert a conflicted transaction into the correct time when resources become available.

Wrapper functions are changed from this. `tryAccessDS()`, `setAccessTimeDS()` are used in the `checkResourceAvailableDS()` to handle various access modes and set resource usage correctly with various access modes.

The `timeToReinsert()` wraps the `getResourceAvailableTime()` method.

Control flow of the Resource Conflict Modeling

In the Ruby memory system, each cache controller has queues for incoming traffic, and triggers its action only when the queues are not empty. The code block with `in_port` keyword is the code that check transactions in the queue and will trigger events to process them. The RCM modeling must be added for each `in_port` block for each queue. We can check the resource availability (bank conflict) with the `checkResourceAvailableDS()` method.

However, we can not use the `checkResourceAvailableDS()` method right after the line of a code for the `in_port` keyword. Depending on the type of messages in the queue, we may need to access a state between the cached state in the TBE and the state from the tag array. The resulting transition can lead to different accesses, for example, only on the TBE, on the tag array only, on the both array. Therefore, the actual bank conflict check is added per an event considering both the current state and the next state. The details depends on the detailed access modeling on the SNUCA cache and the DS cache designs which will be discussed in Section 5.3.2.

If the decision from the `checkResourceAvailableDS()` method indicates that the resource is currently in use (bank conflict), the transaction that are currently inspected by the `in_port` code block must be stalled until the resource becomes available again. For convenience, let's call the mechanism that stalls a transaction a blocking mechanism.

About the blocking mechanism

The Ruby memory system provides various blocking mechanisms related to performance efficiency. Before start discussion, I want to introduce a terminology, "action" that is a special keyword of the SLICC. An action is a a single step of overall operations that must

be completed to make a transition into a new coherence state. A set of actions can be used to describe whole operations before changing a state. The following list shows the blocking mechanisms which are provided with the Ruby memory system for a cache memory controller.

Stall. This mechanism is implemented with a special action called `z_stall`. It simply returns to the caller without further inspection in the working queue.

Recycle. This mechanism pops a current transaction from the queue and re-insert the transaction with updated arrival time.

Stall and Wait. This mechanism pops a current transaction and move into a special queue for the blocked packets.

Because each queue is inspected when it is not empty, "Stall and Wait" is the most performance efficient while "Recycle" is the least performance efficient. It seems any one of the blocking mechanisms works fine functionally, but there are some complications which must be handled correctly. The first issue is with sub-banking. If we select the "Stall" mechanism, other transactions in the queue are left unevaluated. This can be a problem when we want to test with sub-banking configuration. Because transactions on the idle sub-bank can proceed, it is possible for some transactions to be blocked incorrectly. Possible solution is to make separate queues for all the sub-banks. However, because a transaction from the L1 cache to the L2 cache is put into a queue from the L1 cache controller, this work-around puts a burden on the performance of the simulator with the increased number of queues.

The second issue is that a blocking mechanism already being used in the L2 cache SLICC codes. "Stall and wait" mechanism is used for the cache replacement. Otherwise, because subsequent transactions on the same set will stay in the queue until the write-back for a victim is done, significant performance overhead can be added from the blocked transaction

being evaluated every cycle. We can select the "Stall and wait" mechanism for the blocking mechanism of bank-conflict modeling for the best performance efficiency, but the implementation would be very difficult. Blocked transactions moved into a separate queue must be returned when the cause for the blocking goes away. In case of cache replacement, when to move back the blocked transactions is obvious. We can reinsert blocked transaction into the original queue when a notification of completion of the write-back is arrived from the main memory. However, there are no obvious events when the resource is released (when the previous access is done).

From the above reasons, I selected "Recycle" to implement the blocking mechanism to model bank conflict. When a transaction is blocked because the sub-bank is in use, a blocking event is raised to trigger recycling action. A blocked transaction must be re-inserted into the same queue, so I prepared 3 blocking events for each incoming traffic queue. During the transition from the blocking event, the `timeToReinsert()` method is called. In the method, the `getResourceAvailableTime()` method is called to get the time when the sub-bank is released from the previous access, and then the blocked transaction is inserted into the time when the sub-bank is released.

There can be a race between the recycled transactions and the transactions in the queue for the blocked transactions from the replacement. It is not an issue with the Data Shepherding cache because there are no generic cache replacement, but with the SNUCA cache, this can be a problem because the transactions from the blocking queue can be inserted after the recycled transactions. This problem can be seen as an order inversion. We can resolve it by keeping orders on the related transactions. If a new transaction tries to access a set where a cache replacement is, it must be moved to the blocking queue. Otherwise, we can recycle the transaction.

State	Access Permision	Location
//Base states		
NP	Invalid	N/A
SS	Read only	Cache line
M	Read and write	Cache line
MT	Maybe stale	Cache line
//L2 replacement		
M_I	Busy	TBE
MT_I	Busy	TBE
MCT_I	Busy	TBE
I_I	Busy	TBE
S_I	Busy	TBE
//Transient states fetching data from memory		
ISS	Busy	TBE
IS	Busy	TBE
IM	Busy	TBE
//Blocking states		
SS_MB	Busy	Cache line
MT_MB	Busy	Cache line
M_MB	Busy	Cache line
MT_IIB	Busy	Cache line
MT_IB	Busy	Cache line
MT_SB	Busy	Cache line

Table 5.4: Coherent states of a L2 cache line and its access permission and location.

5.3.2 The Resource Conflict Modeling and access timing

To set the resource (a sub-bank) usage correctly and get time when the resource is released, we need model for the access mode of a cache memory. The access mode is the term used in the CACTI tool [78] to point the accessing pattern on the tag array and the data array. "Normal mode" is the generic access mode on the cache memory. In the set-associative cache, the comparator result at the tag array side will drive multiplexer drivers to select a block from a set read from the data array. There are two other access modes, "Fast mode" and "Sequential mode". They are more inclined to the scratch pad memory and direct-mapped cache designs. "Fast mode" reads out both tag and data arrays at the same time, while

“Sequential mode” reads out the tag array first, then accesses the data array only when the data is valid. The former is more performance oriented, and the latter has power efficiency in mind. The latter is a frequent choice of embedded CPUs, such as ARM processors [61].

As discussed in [85], the set associative cache with “Normal” access mode has more delay on the critical path of execution due to the overhead of comparators, and multiplexors. The difference in the access latency between a bank with “Normal” mode of the SNUCA cache and a bank with “Fast” mode of the Data Shepherding cache will be discussed in Chapter 6, while the discussion in this chapter stays on the how to model the various access mode for the RCM.

In case of “Normal” mode, I assumed both tag and data arrays are in use with each data access. Because the comparator result selects a block, the tag array is set as busy during the data access. Therefore, this mode holds both arrays as not available during access time. Likewise, the time to re-insert recycled (blocked) transaction is the earliest cycle when both arrays are available.

I implemented the “Fast” mode (parallel access mode) similar to the “Normal” model because both tag and data array must be accessed at the same time to read or write data. Both arrays can be independently accessed, so I could allow overlap between tag array only access and data access because a tag access latency is shorter than a data array latency generally. However, there can be a possibility of updating the same tag of the cache line which is currently being read/written out. I could avoid this situation and implement early tag resource release, but I decided that I would leave it (with no overlapping) without further optimization to get baseline performance. The “sequential” mode is not implemented because this mode is not used in this work. However, the power efficiency of the “sequential” mode is very interesting feature of the Data Shepherding if we are inclined to the power efficient design. The “sequential” access mode is left as the future work.

Until now, we only have talked on the access to the arrays in the cache memory. The states which lead to the access to the arrays in the cache memory is described as "cache line" in the location column of Table 5.4. The another access location is the TBE which is a structure to model the functionality of the Miss Status Holding Registers(MSHR). Even though I set the number of L2 TBEs as default of 256 (the reason is discussed in Section 4.3.2), because the actual count of the TBEs are much smaller (generally implemented with CAM), I set the latency as 1 cycle.

The states which are cached at the TBE are temporal states to improve performance by releasing the cache array to service other transactions. Because the location of the cache line which is requested from a transaction can different, the RCM must be used selectively. For example, when we empty a TBE from a event triggered from a coherence message, we may drop the data from the TBE or need to write data into the tag and data array. In some cases, state transitions do not change the location of the requested cache line. Therefore, to model bank conflicts properly, we must look up both the cache coherent state and the event triggered from the transactions. There are many states and events at the L2 cache, so I made access mode decision functions to set the end of access time correctly at the arrays to be accessed.

In Table 5.4, there are some of the transient states whose cache lines are located at the cache arrays. They are classified as the blocking states. The blocking states do not have any relationship with the blocking mechanism for the RCM. I borrowed the terminology from the description of the states in the SLICC codes for the L2 cache. The blocking states are entered from the exclusive requests from the L1 cache. There are two reasons I consider the cache lines in blocking states as being located at the cache arrays. First, GEM5 implementation does not cache the blocking states into the TBE. More importantly, different from other transient states whose life expectancy in the TBE is bounded by the access time of the main memory, the bound of expected time in the blocking states are open.

The last thing to discuss in the RCM design is how to model TBE accesses. As discussed above, all the transactions in the queue up until the current time are evaluated. If all the TBE accesses with the same time stamps - arrive at the same time or are recycled by the blocking mechanism - are handled in a single cycle, it will be very unrealistic. Because TBE access consumes a single cycle, instead of adding another class object such as one from the `BankedArray` class, I just recycle all other transactions on the sub-bank after a transaction accessing the TBE, and re-insert them after the cycle.

Chapter 6

Results

Some characteristics of the Data Shepherding cache will be evaluated in two different ways. Physical characteristics and benefits from the use of scratch pad memory will be discussed in the first section of this chapter. Then, detailed analysis on the performance impact of the page flush will be followed in the second section.

6.1 Analysis of physical characteristics

One of significant differences between the Data Shepherding cache and other set associative cache designs is the way how the Data Shepherding cache manages each cache blocks to each bank. Instead of searching for one in a set at the local of a cache bank, the Data Shepherding cache already knows the expected placement of a cache block with the help of mapping table - in my design, bank mappings are integrated with the TLB cache. That's why we can utilize a scratch pad memory for the design of a bank.

To figure out the benefits from the use of a scratch pad memory, I will compare the Data Shepherding cache configuration to the Static Non-Uniform Access (SNUCA) cache memory.

I will explain why SNUCA is chosen to make a comparison in the first sub section. Then, detailed comparisons between the two cache configurations will be made with the CACTI simulation results.

6.1.1 Details of the SNUCA cache

The terminology of Non-Uniform Cache Access (NUCA) is borrowed from [58]. Because the access time of a single bank cache memory - the authors called it as a Uniform Access Cache (UCA) memory - grows fast as the size grows, dividing one into multiple banks would be better in order to achieve better average access time. Different from the UCA cache, there are multiple links connected to the NUCA cache. Generally, each core is connected to a different on-chip router. Therefore, a more complex network topology than simple bus is applied to the design of a NUCA cache. In the literature, another classification is done on the NUCA caches based on the way of searching for a cache block. Static NUCA (SNUCA) cache finds a bank where a searching cache block may reside with a single hashing or a simple indexing. Hit/miss is decided at the controller of the bank after looking up the set which is indexed by the address of the requested cache block. Another class of design with different searching strategy is Dynamic NUCA (DNUCA). DNUCA cache allows dynamic migration of a cache line. So, a cache block is searched for a bank by a bank in a set of banks where the searching cache block might be cached.

The Data Shepherding cache (DS)'s search method is different from the two. The location of each line is statically decided by the bank mapping until the bank mapping is replaced or remapped. So, the DS is different from the DNUCA. However, the bank mapping is decided from the dynamic decision. Therefore, the DS is also different from SNUCA. By the way, the DS cache is a multi-banked cache where each bank is interconnected by on-chip network

and there are multiple access link to the DS cache. We can consider the DS is a NUCA design, and a DS is a different class of design from the DNUCA and SNUCA cache designs.

The reason I selected SNUCA cache for the comparison with the DS cache is because of its current popularity, but also most of DNUCA projects still remains in the domain of research. For example, the shared last level cache of the Intel Haswell chip [46] consists of 4 banks, and each bank is connected to the on-chip ring network. The ring network connects 4 cores, on-chip graphic controller, and a system agent where other system peripherals are attached. As indicated in [73], the last level shared cache is interleaved, a bank to find a cache block is indexed by some bits in its address, which means the method to select a bank is fixed and static. So, we can consider the Intel Haswell chip is an example of SNUCA chips.

There are many parameters which describes physical characteristics of a cache memory, but some important parameters are access latency, area, dynamic access energy, leakage power, etc. The NUCA caches need additional parameters to describe on-chip network such as link/router latency, link/router access energy, link/router leakage power, etc. On-chip network parameters have some relationship to the parameters of a bank. The relationship can be observed more clearly from the view point of the performance. The following inequality is the condition when the performance benefit of the NUCA caches can be claimed over the UCA cache memory. The inequality assumes that accesses to each bank is uniformly distributed. Also, We assume that all the bank in the NUCA cache has the same capacity, and the sum of all the bank in the NUCA cache is the same with the capacity of the UCA cache.

$$Latency_{Network} + Latency_{BankAccess} < Latency_{UCA} \tag{6.1}$$

In Equation 6.1, $Latency_{Network}$ is average latency to any bank in the NUCA cache. Again, $Latency_{Network}$ can be modeled further with Equation 6.2.

$$Latency_{Network} = Counts_{hops} \times Latency_{link} \tag{6.2}$$

In Equation 6.2, $Counts_{hops}$ is average number of hop count to reach out each bank. It is obvious that we can utilize on-chip network to hide some access latency away, which is not doable with the UCA cache because the access time is always the same. If we can change access pattern to the cache memory, it is the same effect of reducing the network latency because of reduced average hop counts. The DNUCA cache achieves the benefit by using distance (hop counts) to each bank as a stack distance in the LRU queue. With hits, a cache block would migrate banks closer to the CPU cores (requesters of the cache block). Cache blocks which are frequently used are naturally are moved closer to the banks so smaller average hop counts can be attained. In the DS cache, page mappings can be made closer to each requester which would build effective virtual partitions for each core. As discussed in Chapter 5, I want to model baseline performance to compare with the SNUCA cache fairly, so selected LRU based scheme for the page replacement.

Even without tweaking average hop counts, still the difference of a bank architecture distinguishes the DS from the SNUCA cache. The area of a scratch pad bank is smaller than a set associative bank. This will give the DS smaller access time to a bank over the SNUCA cache. Smaller area also has implications in other ways. Smaller size helps to have a smaller link size which provides room for performance boost by increasing network frequency. Variable frequency and voltage control schemes are already being used in the chip designs such as the Intel SCC [68]. Other direction of implication is about power consumption. Especially, leakage power is proportional to the area used. This will give a designer more design choices

Parameter	Value/Setting
//Common Settings	
`Cache line size`	64 bytes
`Cache type`	cache
`Technology`	32 nm
`Input/output bus`	512 bits
`Link width`	128 bits
//Individual Settings	
`Tag size (DS)`	7 bits
`Tag size (SNUCA)`	45 bits
`Access mode (DS)`	fast (parallel)
`Access mode (SNUCA)`	normal
`Associativity (DS)`	direct mapped
`Associativity (SNUCA)`	16

Table 6.1: CACTI settings used for experiments.

- better power efficiency for the same capacity over the SNUCA cache, or better performance efficiency by increasing capacity. In the next sub section, we will take a look at the detailed differences and the implications with the result from the CACTI simulator [78].

6.1.2 Comparison between the Data Shepherding and the SNUCA cache configuration

The settings used for CACTI are listed in Table 6.1. Capacity and bank counts are variables, so do not appear in the table. Except them, other parameters not described in the table are left as default values. Cache line size is chosen as 64 bytes because it was popular choice with 32 nm technology CPUs [45, 55]. Input/output bus is internal bus of a bank. Because a line size is 64 bytes, I kept the input/output bus the same. The link width of on-chip network is selected from the default value of GEM5. Other parameters and the values are from the discussions made in Section 4.4.

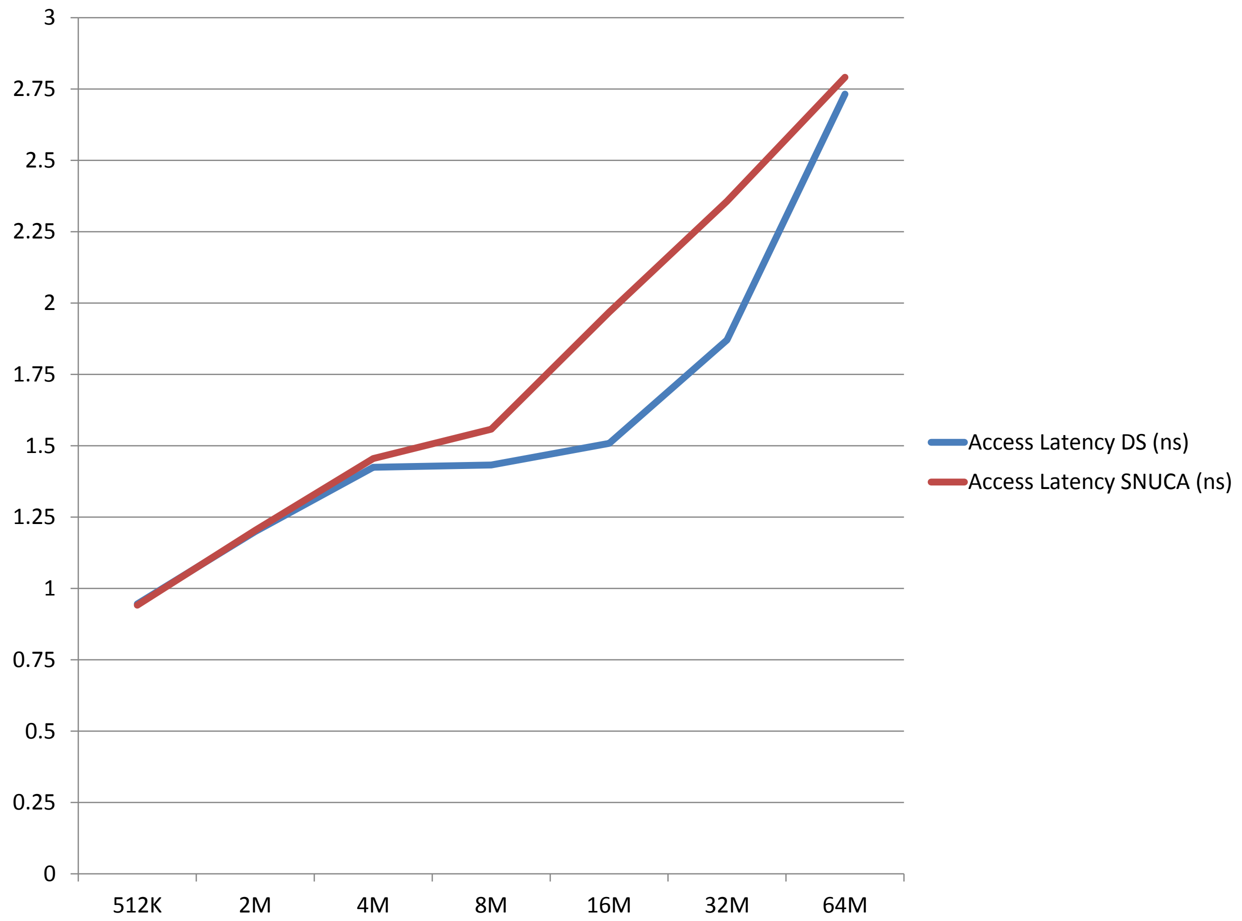

Figure 6.1: Single bank access latency of the Data Shepherding cache and the SNUCA cache.

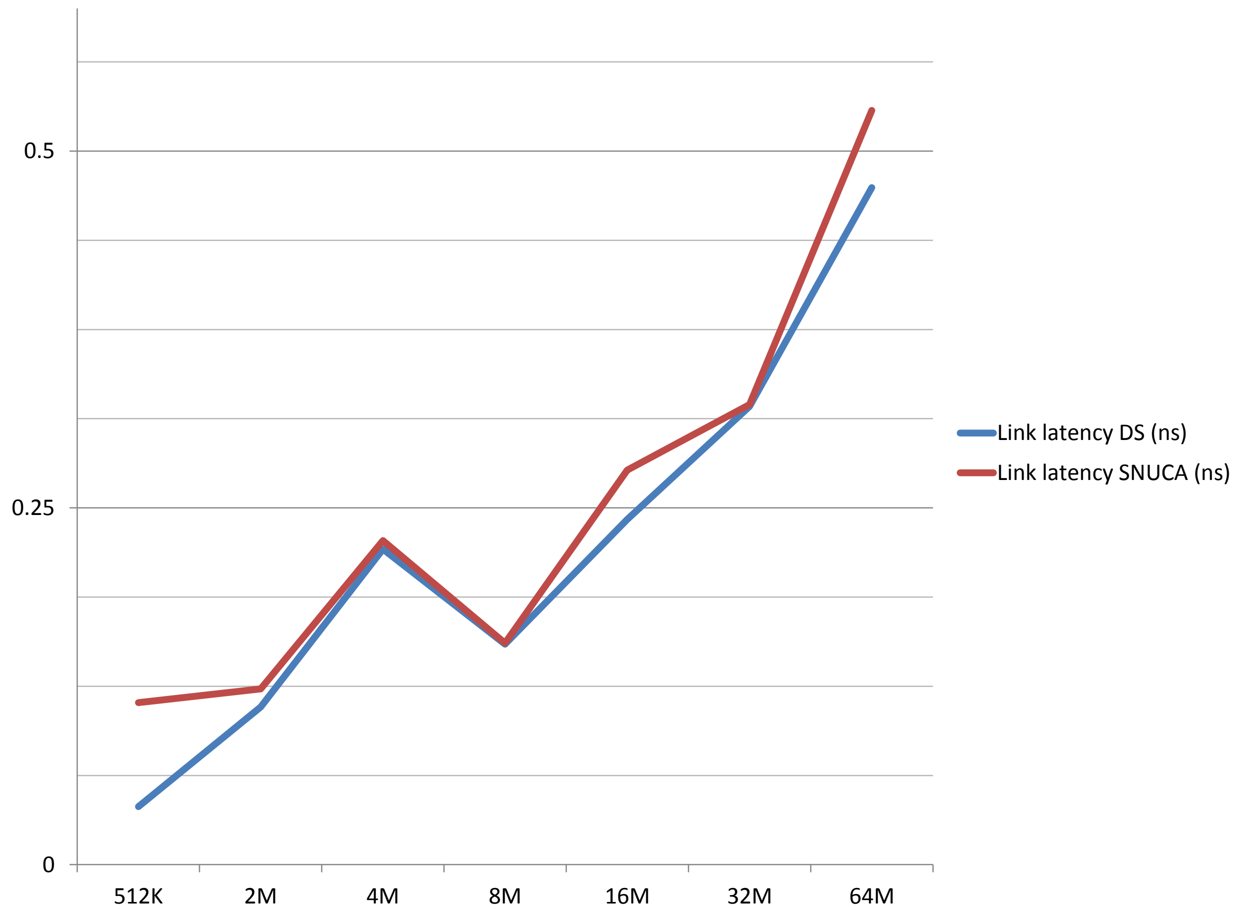

Figure 6.2: Latency of the longer link (between one for x-axis and the other for y-axis) of the Data Shepherding cache and the SNUCA cache.

Node counts	2	4	8	16	32	64
Topology	2 x 1	2 x 2	4 x 2	4 x 4	8 x 2	8 x 8
Average hop counts	1.5	2	3	4	6	8

Table 6.2: Average hop counts on 2D mesh network with varying nodes.

Because of many automatic optimizations on generating cache configuration from the CACTI, the line graph in Figure 6.1 staggers a lot, so it is not easy to read out any strong trend. But still the DS results are always better (faster) than those of the SNUCA. The link latency comparison graph in Figure 6.2 has even zigzag result lines, but it also shows that the DS has always (even though sometimes very marginally better) better link latency.

One thing I want to note here is that because the direct-mapped with 7 bit tag array makes a DS's bank a rectangular with more skewed ratio between two sides than SNUCA's. Sometimes the maximum link latencies are marginally different between the two, but the other link latencies generally much different and the DS's is generally much smaller. Therefore, as we will discuss again in this sub section again, the DS bank access latency only gets marginal latency when we increase the degree of sub-banking because it makes the ratio between the two sides closer to unity.

The performance benefit of NUCA caches over UCA caches is from the average access time. I utilizes Formula 6.1 for analysis. To take hop counts into consideration as Formula 6.2, latencies must be counted in cycles. Here, I assumed 4 GHz operation frequency. In Table 6.2, average hop counts are listed with various 2D mesh topologies. With this, I get average access time in cycles as shown in Table 6.3 which also shows various sized caches from various topologies and average hop counts in Table 6.3.

As discussed, average access time we are discussing now assumes equal accesses to every bank, and only consider the cache itself - do not consider any other system details, etc. So, this is to figure out potential scalability and how it scales. To show the effect of link latency,

Total Size(16M bank)	32 MB	64 MB	128 MB	256 MB	512 MB	1024 MB
	Average access time in cycles					
DS (16M bank)	8	8	9	10	12	14
SNUCA (16M bank)	10	11	13	15	19	23
Total Size(64M bank)	128 MB	256 MB	512 MB	1024 MB	2048 MB	4096 MB
	Average access time in cycles					
DS (64M bank)	14	15	17	19	23	27
SNUCA (64M bank)	17	18	21	24	30	36

Table 6.3: Selected average access time of the Data Shepherding cache and the SNUCA when 4 GHz operation frequency is assumed.

I selected 16 MB and 64 MB bank sizes. From Figure 6.1, 16 MB bank of the DS cache has 6 cycles, and that of the SNUCA cache has 7 cycles with 4 GHz operation frequency. 64 MB bank of the DS cache has 11 cycles latency while 64 MB bank of the SNUCA cache has 12 cycles. I made another assumption with link latency. In Figure 6.2, the link latency of 16 MB bank of the DS cache can be clocked with 4 GHz, so each link is counted as 1 cycle. On the while, the latencies of 64 MB bank of the DS and 16 MB bank of the SNUCA cache fall between 0.25 ns and 0.5 ns. As base frequency is 4 GHz, 2 cycles are counted for a link of the two cases. Because the link latency of 64 MB bank of the SNUCA cache is between 0.25 ns and 0.75 ns, I assume its latency is 3 cycles.

Table 6.3 can be read efficiently with the access latency of a 128 MB single bank cache (UCA cache), 5.18504 ns which is 21 cycles under 4 GHz operation frequency. The DS cache with 16 MB bank does not have scaling issue at all throughout the tested cache sizes while keeping average access time below 21 cycles. The DS cache with 64 MB stays good up to 1024 MB. On contrast, the SNUCA caches meet faster the UCA latency. 16 MB bank SNUCA cache stays good until 1024 MB, and 64 MB one holds its efficiency until 512 MB. Even though I selected cases to show the differences, but even though both of the DS cache and SNUCA cache have the same cycles of link latency, still the DS caches have better

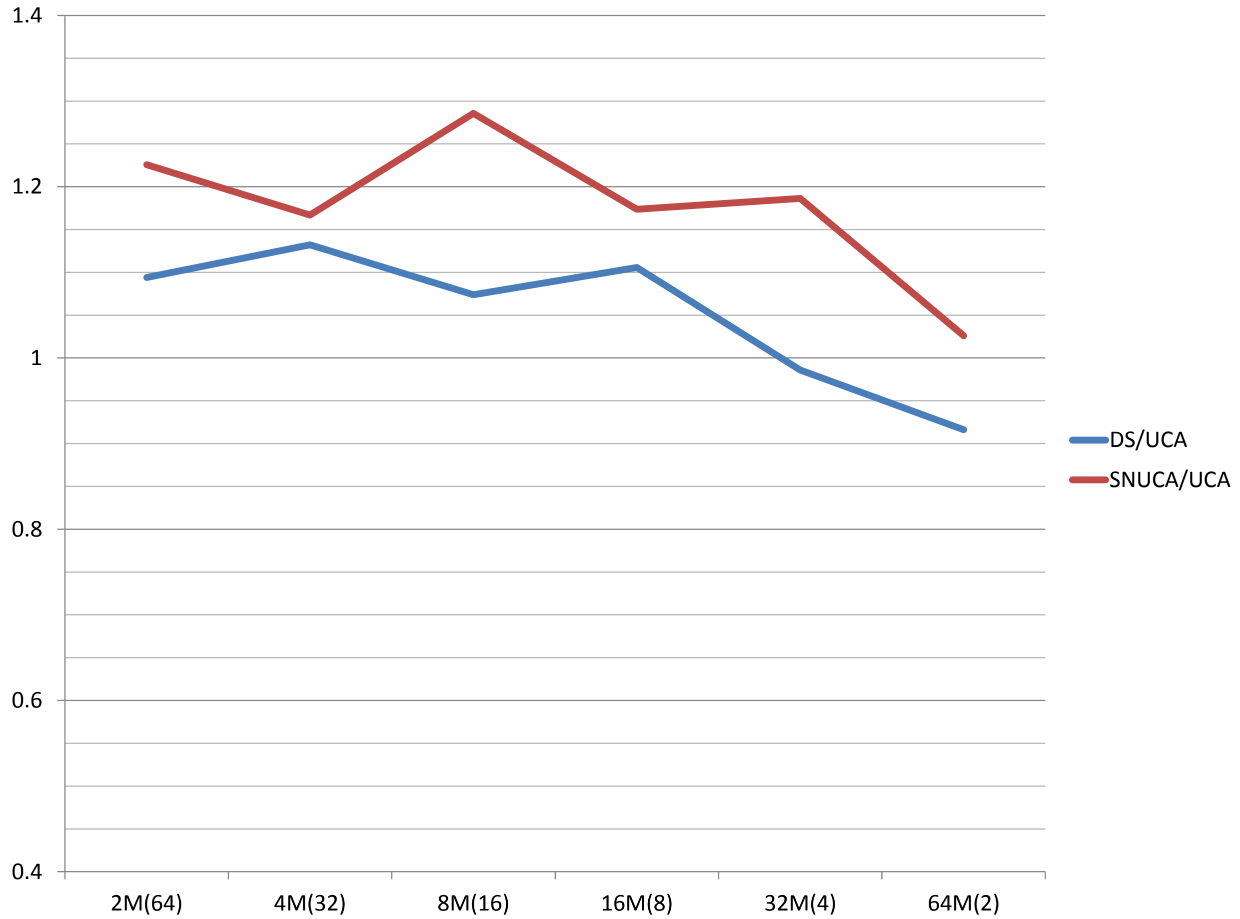

Figure 6.3: Chip area comparison with ratio over the UCA cache. Each cache is sized as 128 MB.

bank latencies, which give more headroom for the scalability. The reason comes from the speed of scaling. As in Table 6.3, average latency increases linearly while the size scales exponentially. In conclusion, even though the differences in the link latencies and bank access latencies look marginal, but they are quite sensitive to the scaling of the cache size. So, the Data Shepherding cache has obvious benefit over the SNUCA cache in this point.

Let's turn our attention into the leakage power and area. Figure 6.3 shows line graphs which are ratios of the size of DS cache over that of UCA cache (128 MB) and the size of SNUCA over that of UCA. The both lines are staggering, but show trend of decreasing as the number of banks decreases. The DS shows better area efficiency as the graph line always stays under

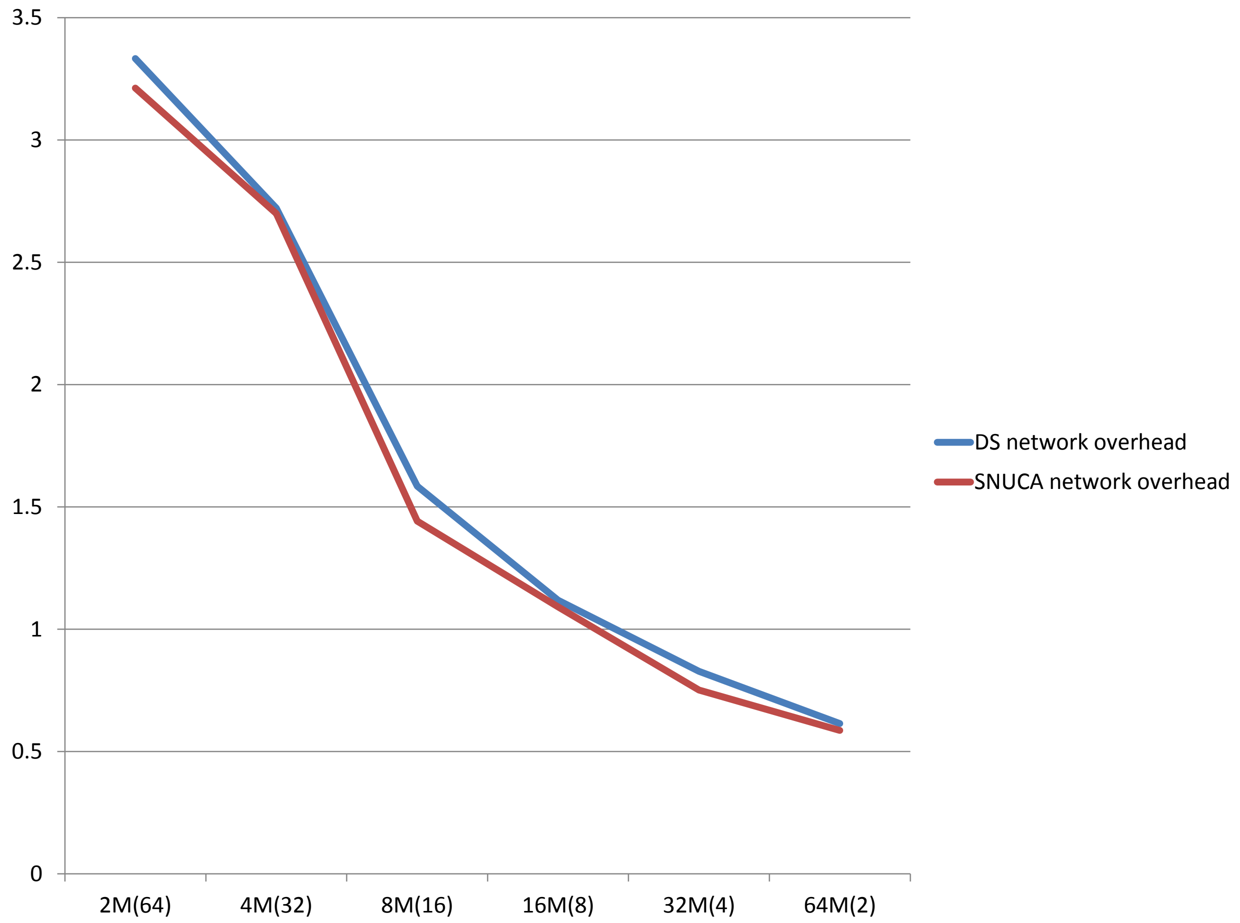

Figure 6.4: Percentage of on-chip network overhead over total area.

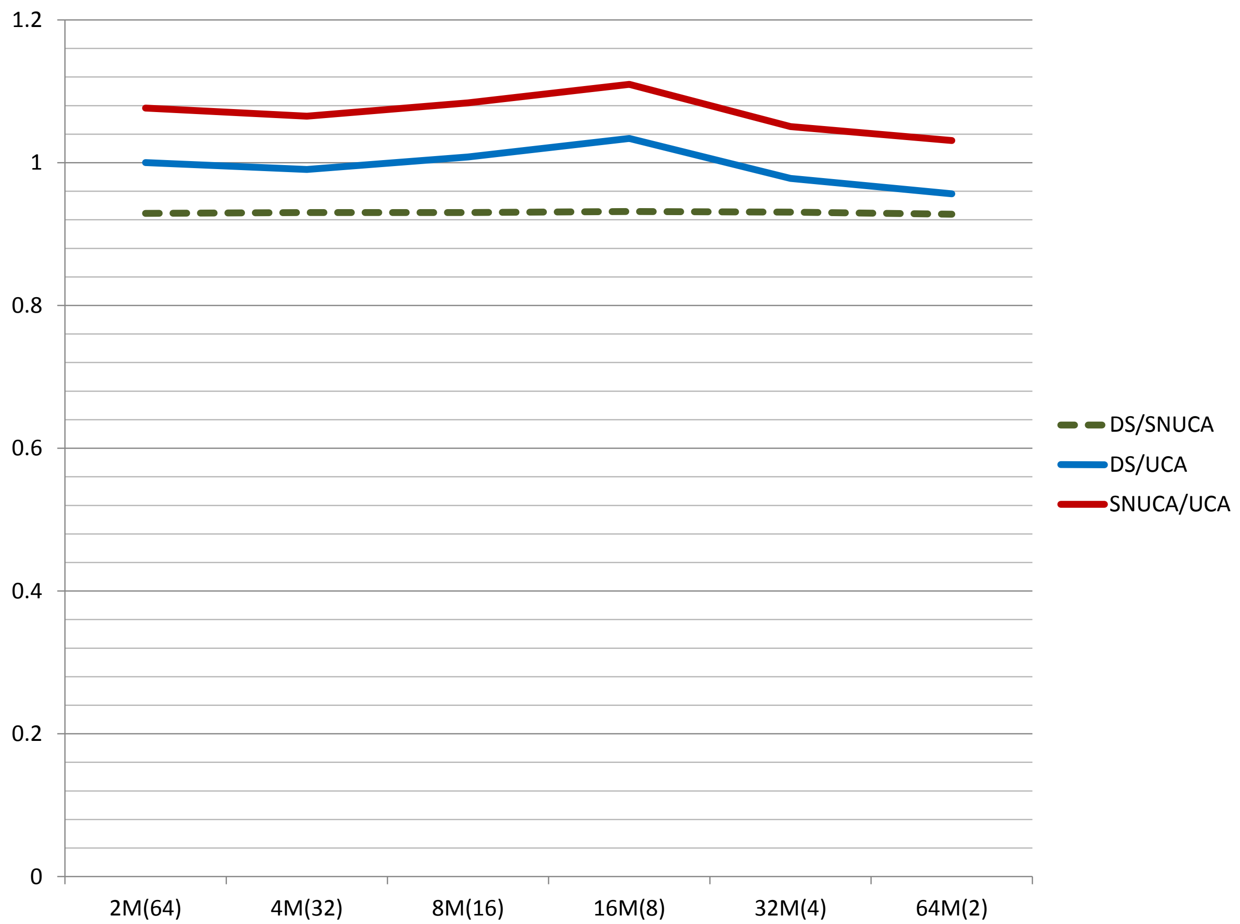

Figure 6.5: Leakage power comparisons with sum of bank leakage power over the 128 MB UCA cache and the DS cache over the SNUCA cache.

that of the SNUCA. In NUCA caches, if a fixed total cache size is given, as we increase the bank counts, each bank diminishes in size and the network area increases. Therefore, the total area increases as with more banks because of network area overhead. The network area overhead in percentage is shown with line graphs in Figure 6.4. As hop counts increases, the area overhead increases linearly. However, the percentage of the network area is relatively small. This is the known benefits of the NUCA caches.

Figure 6.5 depicts graphs of leakage power ratio. Because the leakage power of on-chip network is not provided directly from the CACTI, I took rather convenient approach. The leakage ratios are from the sum of every bank (total 128 MB) over the 128 MB UCA. From Figure 6.4, the network area of the DS and SNUCA caches are roughly identical, so we can still conclude that the DS cache has better leakage power efficiency over the SNUCA cache. As we can see from the green dotted line graph, the DS cache produces less leakage power around 7% than the SNUCA cache.

Other implications from the smaller area of the DS cache over the SNUCA caches is bigger capacity for extra functionalities. For example, the access latency of 2 MB DS bank is 1.19965 ns. When the bank is sub-banked with degree of 2, the access latency becomes 1.08803 ns. In case of the 2 MB SNUCA bank, the latency without sub-banking is 1.20409 ns while the latency with 2 sub-banks is 1.22645 ns. Surprisingly, the DS bank's latency decreases with increased sub-banking. The strange result of reduced access time with increased area comes from the CACTI optimization process. The ratio of the 2 adjacent sides of the 2 MB DS bank becomes closer to unity with increased sub-banking. The 2 MB bank dimension changes from 1.40535 mm x 3.00304 mm to 2.60528 mm x 1.86456 mm.

Parameter	Value/Setting
//Common Settings	
Instruction set architecture	x86
Cache system model	ruby system, directory based mesi protocol
System clock frequency	4 GHz
Main memory size	16 GB
Main memory model	DDR-400
Cache line size	64 Byte
Network model	simple network
Max outstanding requests	16
Number of TBEs	256
L1 cache size	32 KB (I-cache), 32 KB (D-cache)
Bandwidth factor	16 bytes (128 bits)
Warm up period	2 billion instructions
Simulation period	400 million cycles
//Individual Settings	
Associativity (SNUCA)	16
Page replacement policy (DS)	Lest Recently Used (LRU)
Cache replacement policy (SNUCA)	LRU
Number of FSBs (DS)	1
Number of PFTEs (DS)	1

Table 6.4: GEM5 settings used for experiments.

6.2 Performance analysis

In the previous section, we did Average access time analysis using CACTI, and observed that the Data Shepherding (DS) cache performs better than the SNUCA cache if total accesses were uniformly distributed and the same. However, as discussed in Chapter 4, the DS cache design has extra performance from the page flush process. We will evaluate the performance impact of the page flush process with GEM5 simulator [25].

6.2.1 Experiment setup

Table 6.4 summarizes simulation settings which are different from the default settings and some notable settings. Settings related to the shared L2 cache are not listed because they are variables of experiments. The L2 cache also has 64 Byte of cache block size.

Some parameters are selected large enough not to add significant influence on performance. For example, "Max outstanding requests" and "Number of TBEs" are selected with large size as discussed in Chapter 5. Network model I selected is the default of ruby memory system, "simple network." It models on-chip routers with a very simple model, and assumes infinite buffering. Also, routing algorithm finds shortest routes available. Network link is modeled with "bandwidth factor." It is translated into a link width. Default value is 16 bytes, which is equal to 128 bits link width.

The workloads for the experiment are from Spec 2000 suite [6]. Because of limited time, we selected some benches instead of running all. The selected are `crafty`, `gcc`, `gzip`, `mcf`, `twolf`, and `equake`. We tried to select ones with different characteristic. `equake` is from the float point component of Spec 2000. It is to simulate seismic wave propagation. All others are selected from integer components of the Spec 2000. `crafty` is emulating playing chess. So, we can expect that this will not read/write a lot from and to the main memory. `twolf` is from Computer Aided Design (CAD) tools. `gcc` is from compiler tools. We expect both will use complex algorithm while read/write more from and to the main memory than `twolf`. `gzip` is compression/decompression program, and `mcf` does combinatorial optimization for vehicle scheduling. The last two benches are expected to read/write a lot more to and from the main memory. All the runs use the reference work sets of the Spec 2000.

To measure performance, we selected 2 billion warm-up time, and then resets counters. And then, let the simulation run for 400 million cycles to collect experimental results. I decided to go to the direction of running during the selected period time to measure performance. This

method is popularly selected because of very long running time when we run a workload fully. As shown in the experiment details of [56, 93], for 4 MB cache, 10 million instructions are used for warm up (after 1 billion fast forwarding), then performance were measured during 100 million instruction. For 16 MB cache, 100 million instructions are used for warm up (after 1 billion fast forwarding), then the simulator was run for another 100 million instruction to measure performance. Both used the same experiment period while the warm-up periods are increased as the size of cache is scaled. I selected 400 million cycles instead of 100 million instructions is first, because I could not set instruction counts after warm-up time with the version of GEM5 I used, and also because Instructions Per a Clock (IPC) of a single issue out-of-order CPU is generally less than 1. By the way, simulation results from GEM5 give the total instructions executed after the counters are reset (after warm-up is done). IPC is calculated from the instruction counts from the simulator divided by 400 million cycles.

Warm-up time is decided experimentally. In case of first mappings when cache is empty, there would be no overhead from page replacement which can influence the average performance overhead of page flushes, so I selected the point when there is no page replacement on the empty page. One note here is that in the work of [56, 93], fast-forwarding option in GEM5 simulator was used, but fast-forwarding option is not available with ruby memory system, so the option was not used the experiments in this work. However, the increased warm-up time covers both warm-up time and fast-forwarding time, so I think 2 billion warm-up is reasonable choice.

The important parameters not listed in Table 6.4 are variables for each run of experiments. A bank size, variables related to topology (total number of banks, number of lows of 2D mesh, etc.), link latency, access latencies (to tag and data arrays in a bank), etc. Please note that the use of separate access time on tag and data array to simulate different access mode between the DS bank and a SNUCA bank. Topology configuration is set as in Table 6.2.

6.2.2 Experiment results and discussion

I started experiments with 2 MB bank size because 2 MB bank size is used in many Intel Chips on the market [45, 46]. Also, as we can see in Figure 6.1 and 6.2, the link latency and bank access latency are quite close, I think 2 MB size is good start point to demonstrate the overhead of page flushes on performance. Both the DS cache and the SNUCA cache have 5 cycles of access time at 4 GHz operation frequency. However, the tag access time varies - the DS has 2 cycles while the SNUCA has 3 cycles. Each Link latency is counted as 1 cycle as we discussed in the previous sub section. To see there are any benefits with large sized cache, I started experiment with 128 MB. Network configuration is 8 by 8, 2D mesh topology.

Before starting discussion, I want to mention the significance of the cache size on the Spec 2000 benches. Memory footage of the Spec 2000 suite is less than 200 MB as we can see at [7] because it was designed late 90's when 256 MB of main memory is equipped for high-end servers. Most of benches in the Spec 2000 runs well under less than 100 MB of main memory. Therefore, with the enough warm-up time, it is possible to see perfect caching of all workload onto the cache memory because of the large size.

Figure 6.6 shows IPC of some selected benches from Spec 2000. The DS cache shows 12.6% performance improvement on average over the SNUCA cache with the selected runs. Due to the large size, it is very rare to see page replacements from the DS cache and cache replacements from the SNUCA cache. Only `gzip` result reported 416 cache replacements on the SNUCA cache while 104 page replacements with the DS cache. Except `gzip`, we can assume the performance benefits should come from the tag access latency and bank conflicts. The additional analysis is in Figure 6.8 and 6.7.

Figure 6.7 gives some explanation on the DS cache's performance edge over the SNUCA cache. Throughout the results from benches, the DS cache shows less bank conflicts. It

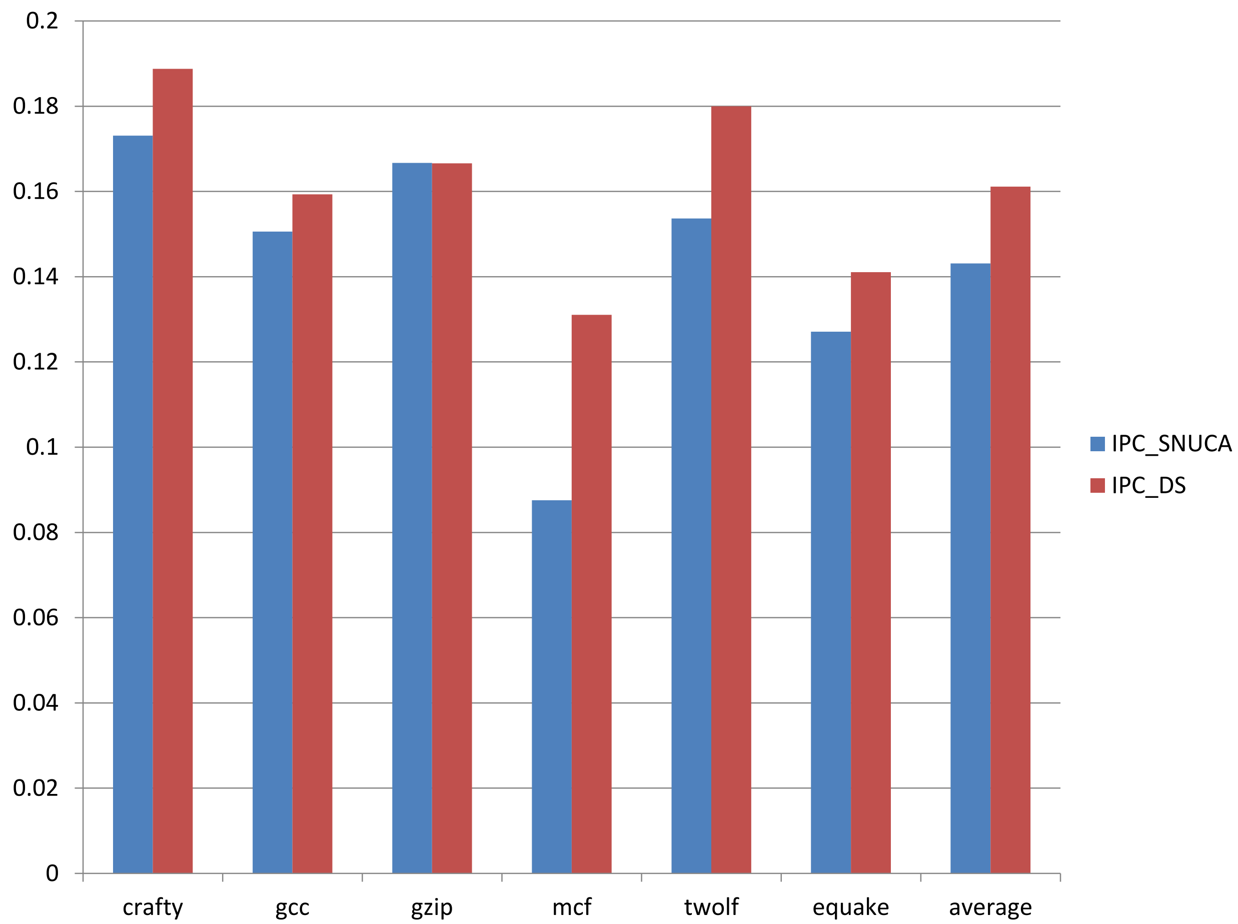

Figure 6.6: Performance comparison between the DS and the SNUCA cache. 2 MB bank size, 128 MB in total.

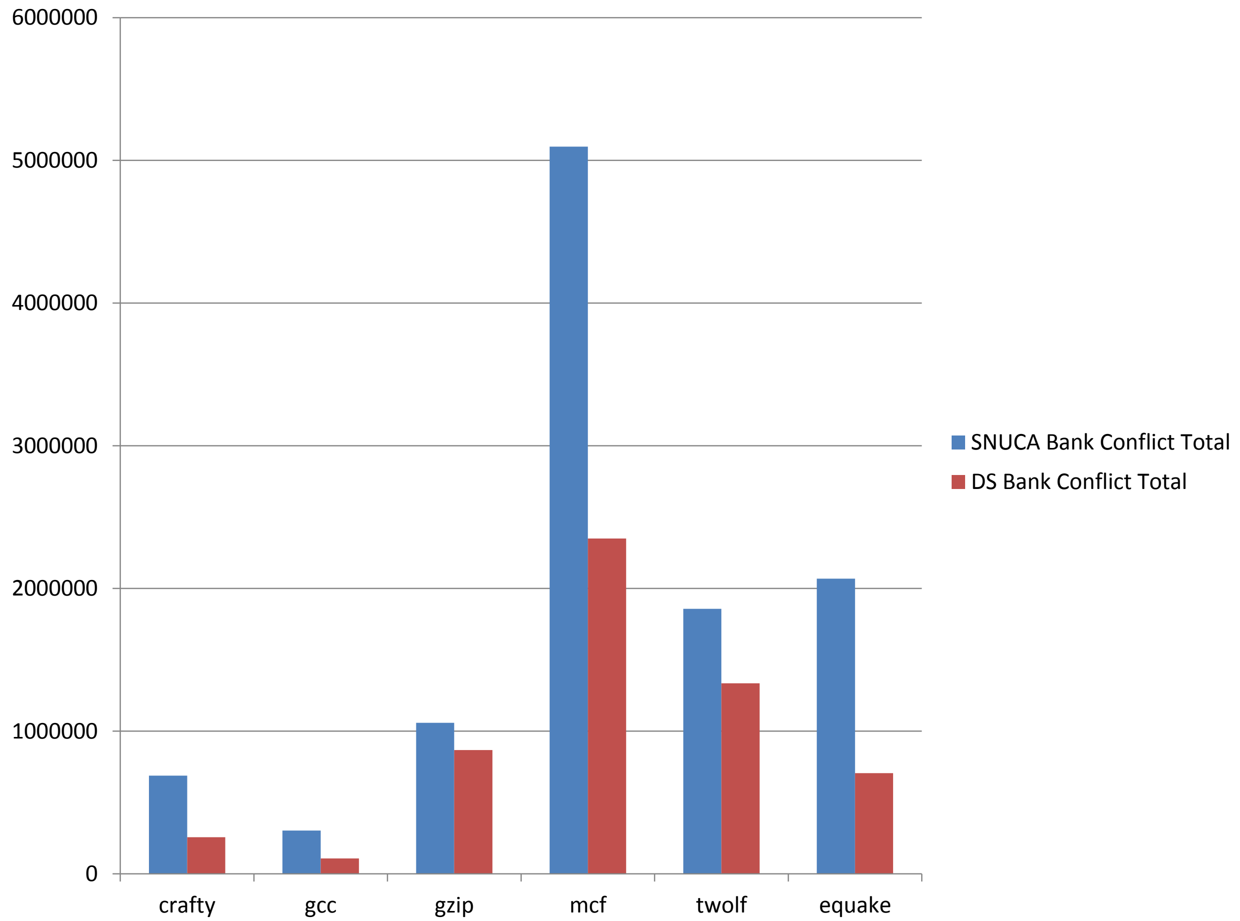

Figure 6.7: The total counts of bank conflicts. 2 MB bank size, 128 MB in total.

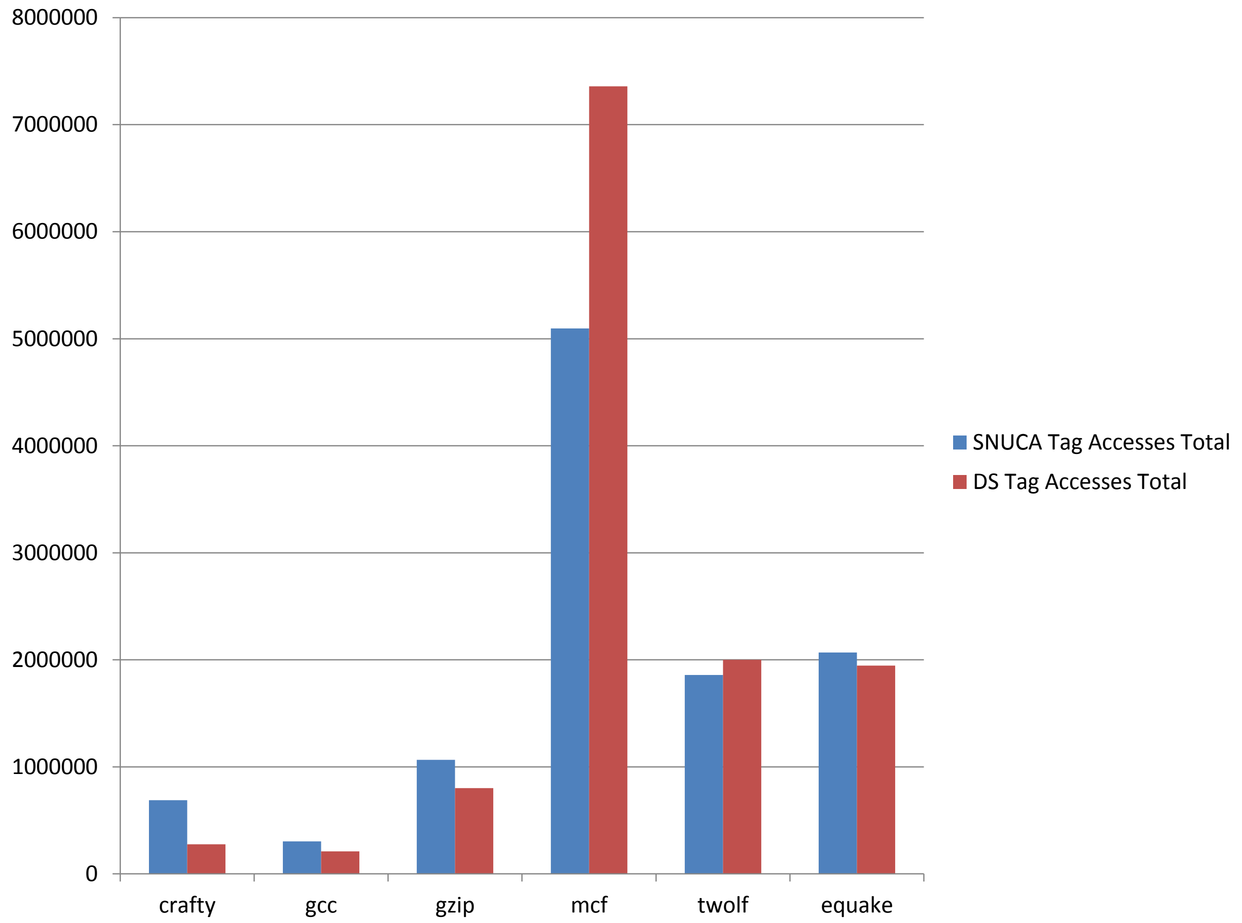

Figure 6.8: The total counts of tag latency accesses. 2 MB bank size, 128 MB in total.

is quite different from what we expected. The DS cache was expected to have more bank conflicts because a whole page is mapped to a single bank. Generally, interleaved cache design is considered better in order to reduce the counts of bank conflicts. So, I looked for the possible explanation. Figure 6.8 has some notable points which can be clues.

Figure 6.8 depicts the counts of tag latency accesses from each run of benches. As we discussed in Section 5.3, there are accesses which only hold resources during tag access time. For example, hit/miss decision is known to the cache controller after the tag access time. Another example is coherence updates when a block is not cached at the TBE (functions as the MSHR in GEM5). `crafty`, `gcc`, `gzip`'s tag latency accesses are quite similar to the counts of bank conflicts. Further inspection shows that all of the tag latency accesses are from the "exclusive unblock" events which cause transition from the Exclusive states to other coherence states. The reason for the state change in a single threaded run is because of "write" on the cache blocks which can not be avoided from the interleaved accesses. `mcf`, `twolf`, and `equake`'s results are different from the first three cases. The numbers from DS cache's tag latency accesses are way more than those of the DS cache's bank conflicts. Still, the SNUCA cache shows similar trend as the first three cases. What makes the significant reduction in the bank conflicts from the DS cache? Possible explanation is that the smaller tag array latency reduces the window of conflict efficiently.

The smaller counts of bank conflicts over that of the SNUCA cache with `gcc` was not carried into performance benefit. It is the only one with page flushes in the first batch of experiments. To look into the effects of page flushes on the performance, we can try again the same bench set with smaller bank counts (smaller total size). Without change on the bank design (2 MB sized bank), we will use total 8 MB, 2 by 2 network configuration. The performance comparison is depicted in Figure 6.9.

Total size of 8 MB was big cache size in late '90s. However, the shared cache size is much smaller than that of memory footage when Spec 2000 suites are run. So, one could expect

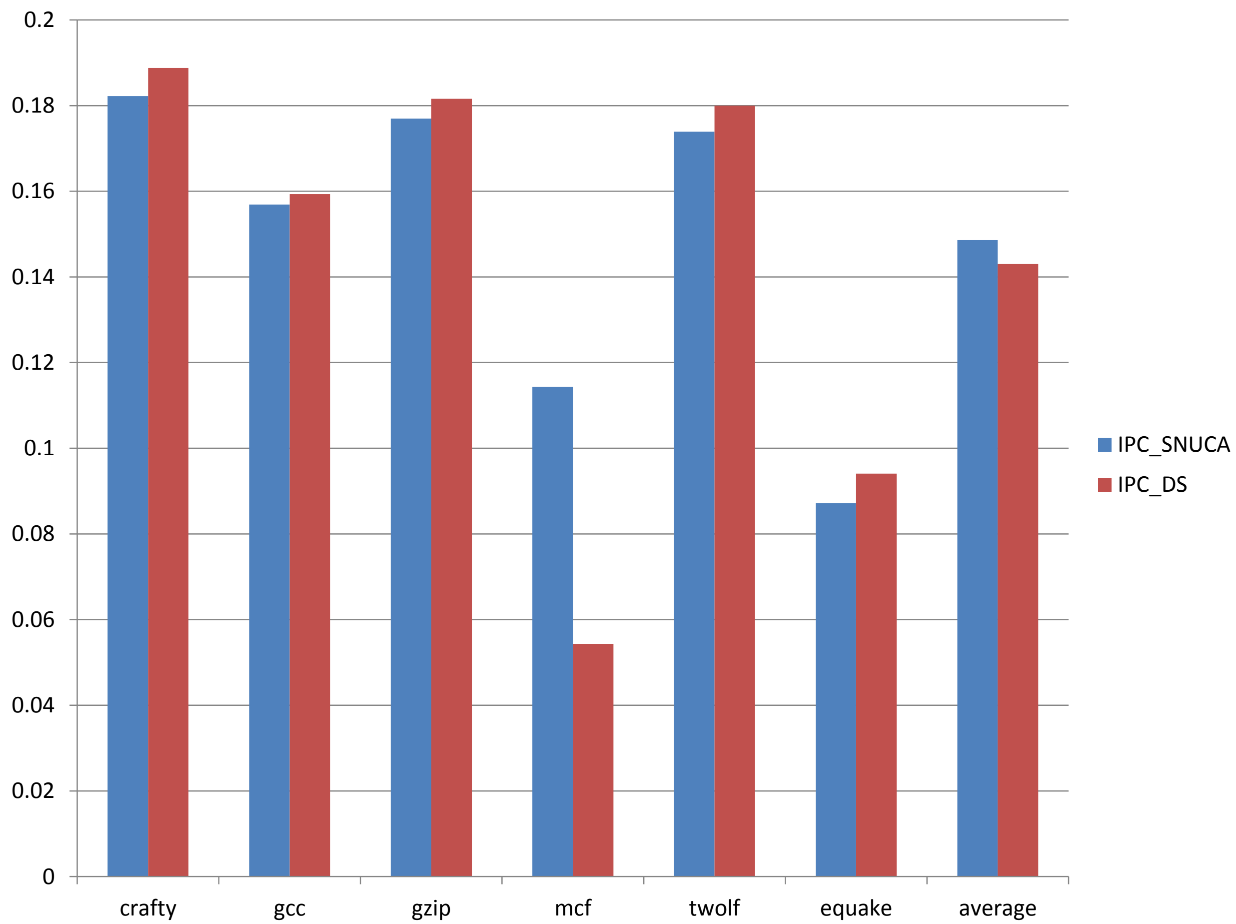

Figure 6.9: Performance comparison between the DS and the SNUCA cache. 2 MB bank size, 8 MB in total.

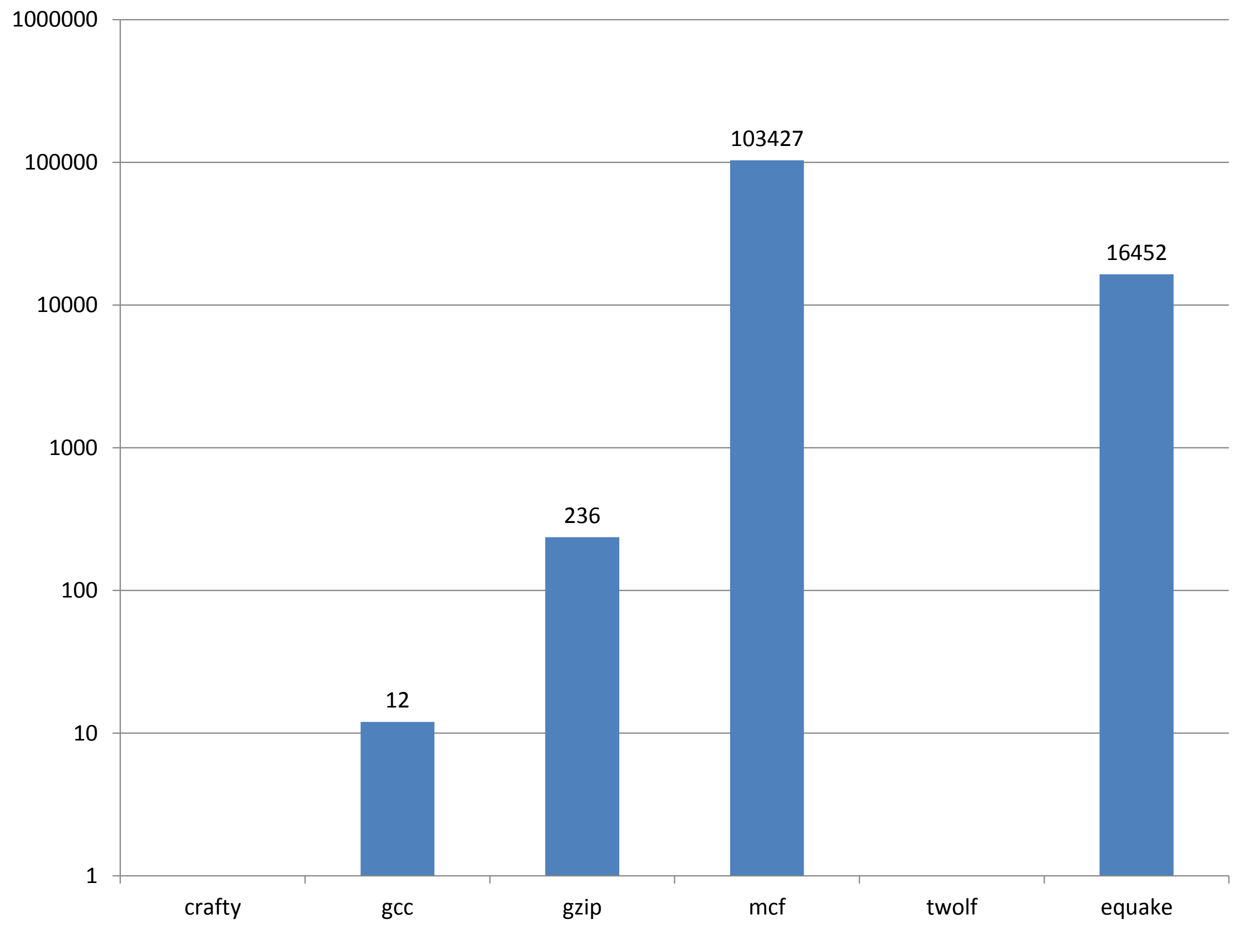

Figure 6.10: The number of page flushes. 2 MB bank size, 8 MB in total.

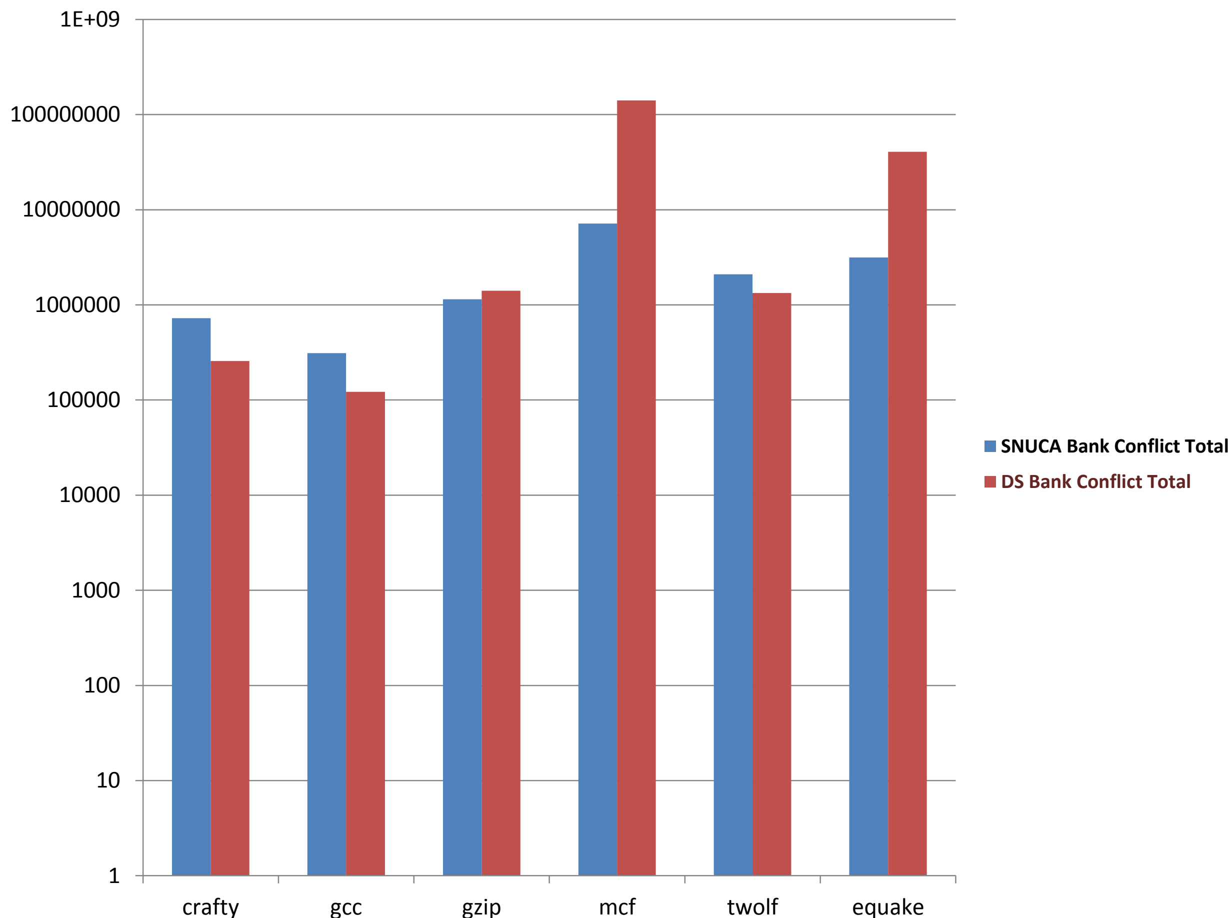

Figure 6.11: The total counts of bank conflicts. 2 MB bank size, 8 MB in total.

that the result from the SNUCA cache would perform better than that of the DS cache. Interestingly enough, the performance results from the DS cache are better in many cases than those of the SNUCA cache. The average is around 4% behind, but except the `mcf` result, the DS cache performance 3.4% better than the SNUCA cache. Still too big to observe page replacements? Figure 6.10 has page flush counts from each spec bench run. The difference of results is surprisingly large enough, so I used log scale for better visibility. `mcf` result shows really huge amount of page replacements. However, `equake` also has significant count of page flushes, but the performance is still better than that of the SNUCA cache. The page replacements from `gzip` are doubled, but the performance is better now.

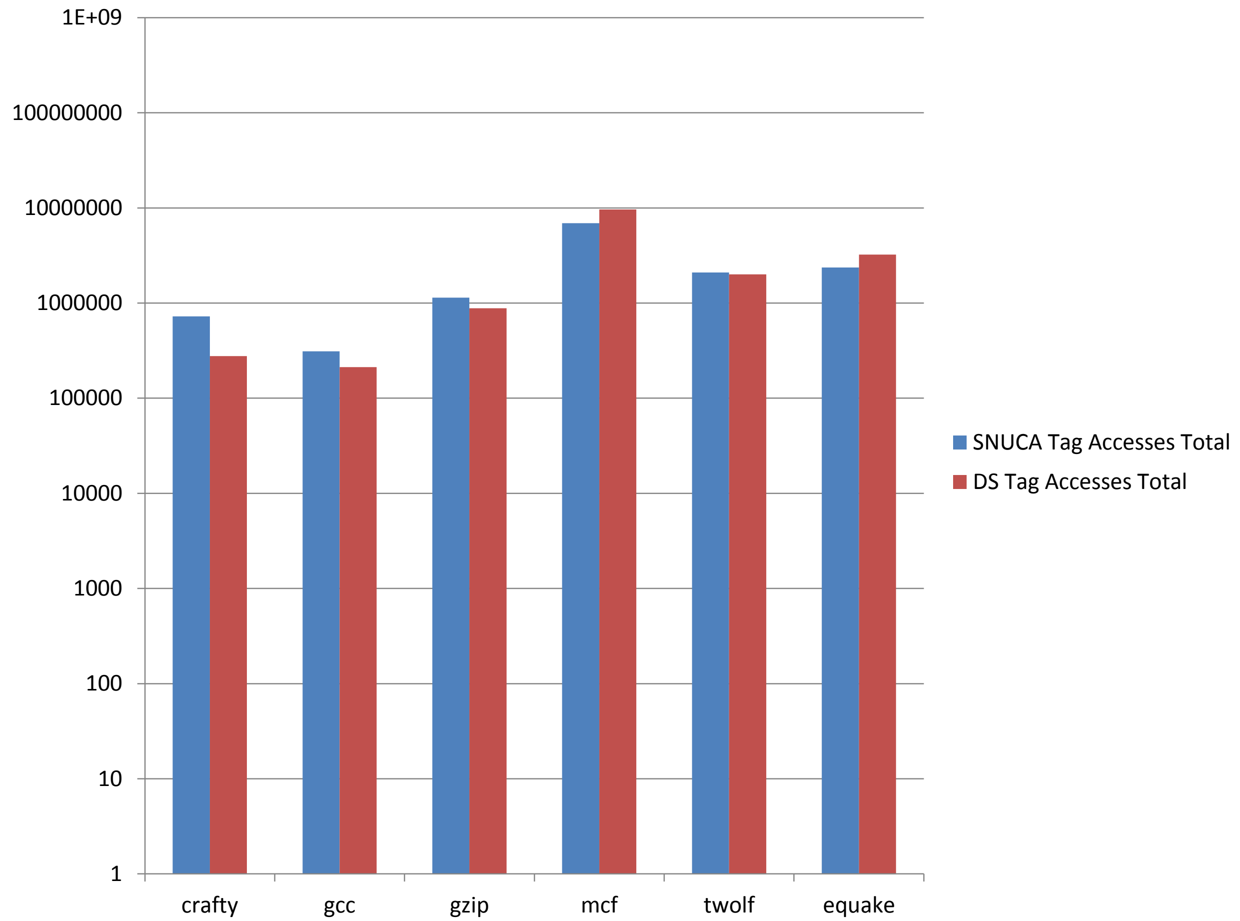

Figure 6.12: The total counts of tag latency accesses. 2 MB bank size, 8 MB in total.

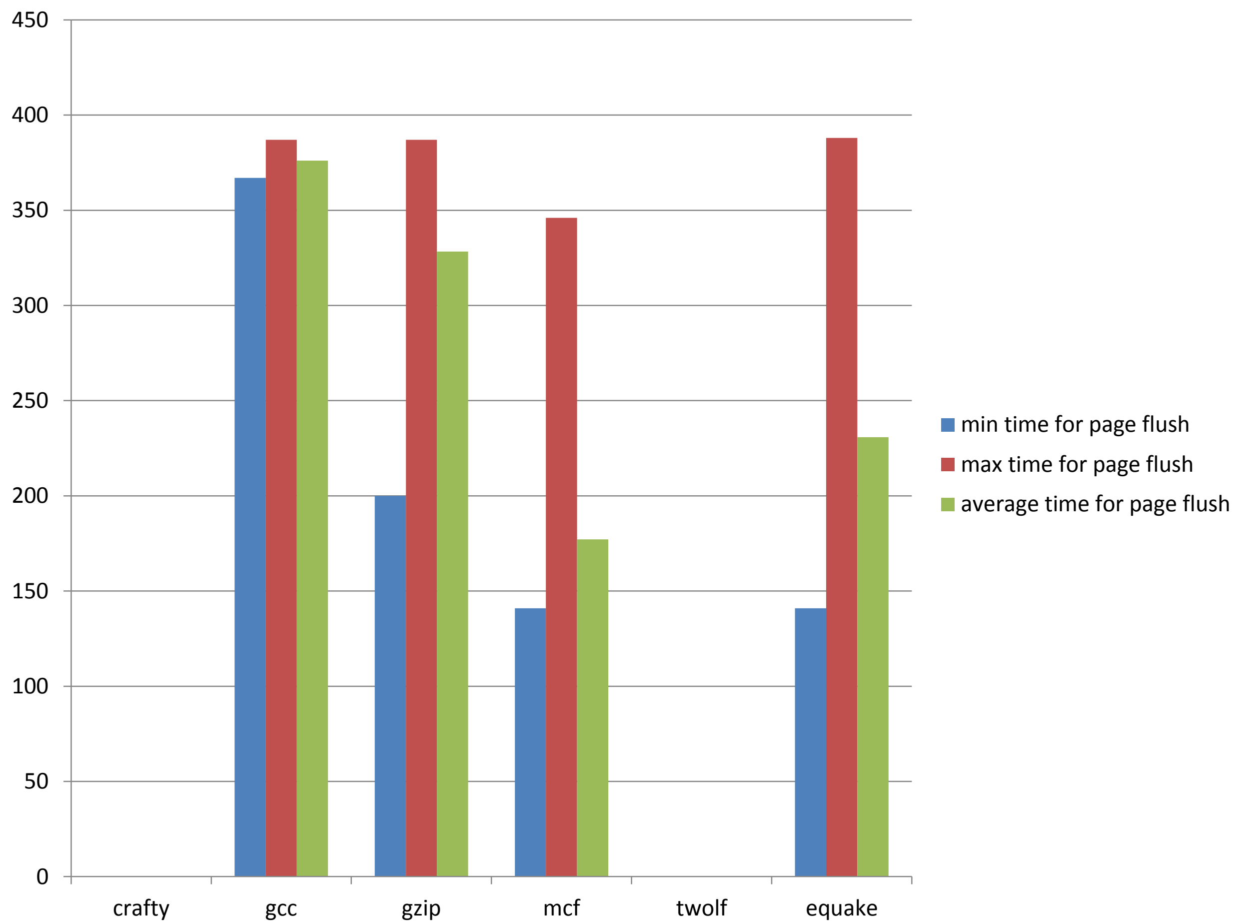

Figure 6.13: Maximum, minimum and average page flush time comparison. 2 MB bank size, 8 MB in total.

As we did with 128 MB runs, let's examine bank conflicts and tag latency accesses. The workloads with no or small counts of page replacements such as `crafty`, `gcc`, `twolf` still have similar counts of bank conflicts. However, the results from `gzip`, `equake`, and `mcf` are quite different. The DS cache has more bank conflicts than tag latency accesses with `gzip`. The `mcf` and `equake` bank conflict counts of the DS cache are significantly more than that of tag latency accesses. So, when more page replacements, the tag latency accesses do not show directly relationship with bank conflicts. One reason for significantly increased count of bank conflicts of `mcf` and `equake` is due to the resource conflict mechanisms. As discussed in Section 5.3, the conflicted accesses are recycled to the back of the queue, so if many smaller time accesses (for example, tag latency accesses) are conflicting with a longer time one (for example, parallel access or normal access) which is under service, smaller ones can make serious of bank conflicts, and every recycling is counted as bank conflict.

Figure 6.13 shows more insight to explain the different trend of performance. Maximum, minimum, and average cycles for a page flush is depicted in the figure. `mcf`'s maximum time is smaller than other's, and also the minimum is lowest. The average is the closest to the minimum except the `gcc`'s case where maximum, minimum, and average are very tightly close. This explains that `mcf` flushes out pages which are only partly used in many cases. Still need more experiments, but this can be explained with Formula 4.1. If a page is fully used on average, this can be considered cache replacement is done in bulk in the view point of performance. As smaller part of a page is used and then flushed out, page flush overhead becomes more significant because the implementation in this work does page flush equally to every page - issues 64 line flushes per a page flush.

To see the influence of increased bank counts, this time we will set the total size the same while a bank size is halved to 512 KB. Network topology becomes 4 by 4, and average network hop counts is increased to 4 from 2. However, because of decreased capacity of a bank, the access latency is now 4, and tag access latency is 2 both for the DS and SNUCA

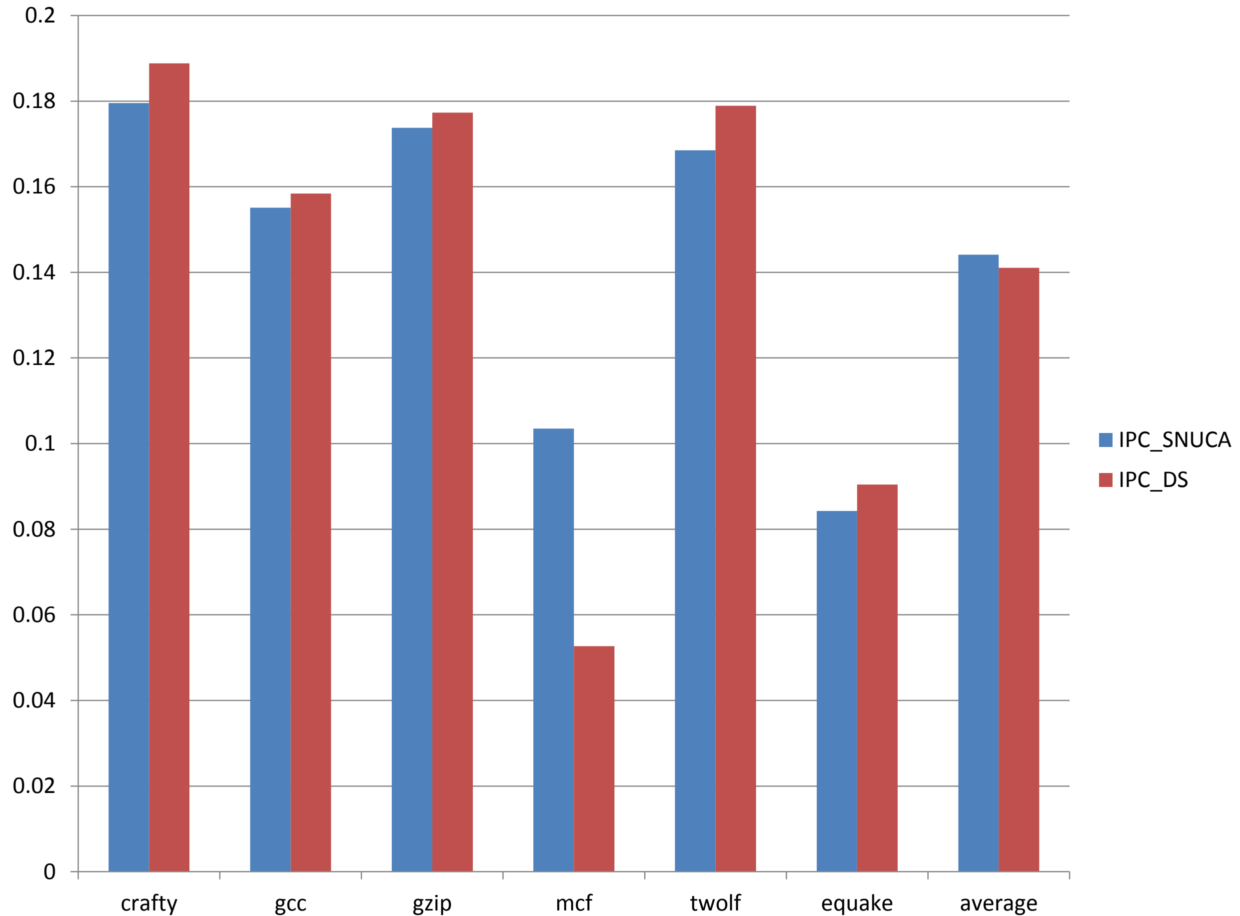

Figure 6.14: Performance comparison between the DS and the SNUCA cache. 512 KB bank size, 8 MB in total.

cache when we assume 4 GHz. Each link latency is set to 1 cycle under 4 GHz from Figure 6.2. Bank latencies are identical, so it will be another good example to examine the impact of page flush and network configuration.

The performance comparison is depicted in Figure 6.14. Every result show a slight performance over 2 MB bank cases in Figure 6.9. Interestingly, some results decreases less than the SNUCA results, so the DS performs 2.1% behind on average which becomes smaller than 2 MB bank case. Also, except `mcf` result, the DS outperforms the SNUCA cache by 4.3% on average.

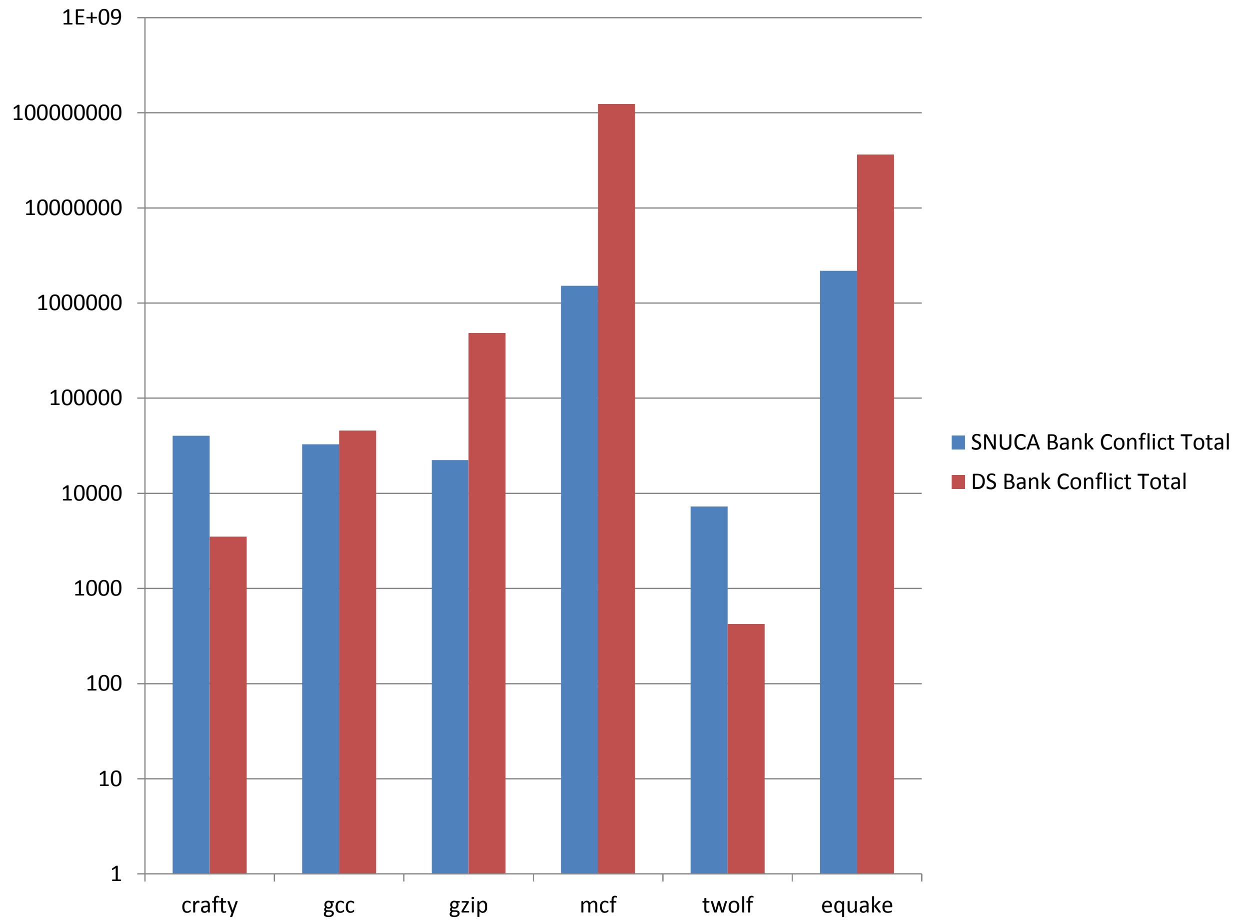

Figure 6.15: The total counts of bank conflicts. 512 KB bank size, 8 MB in total.

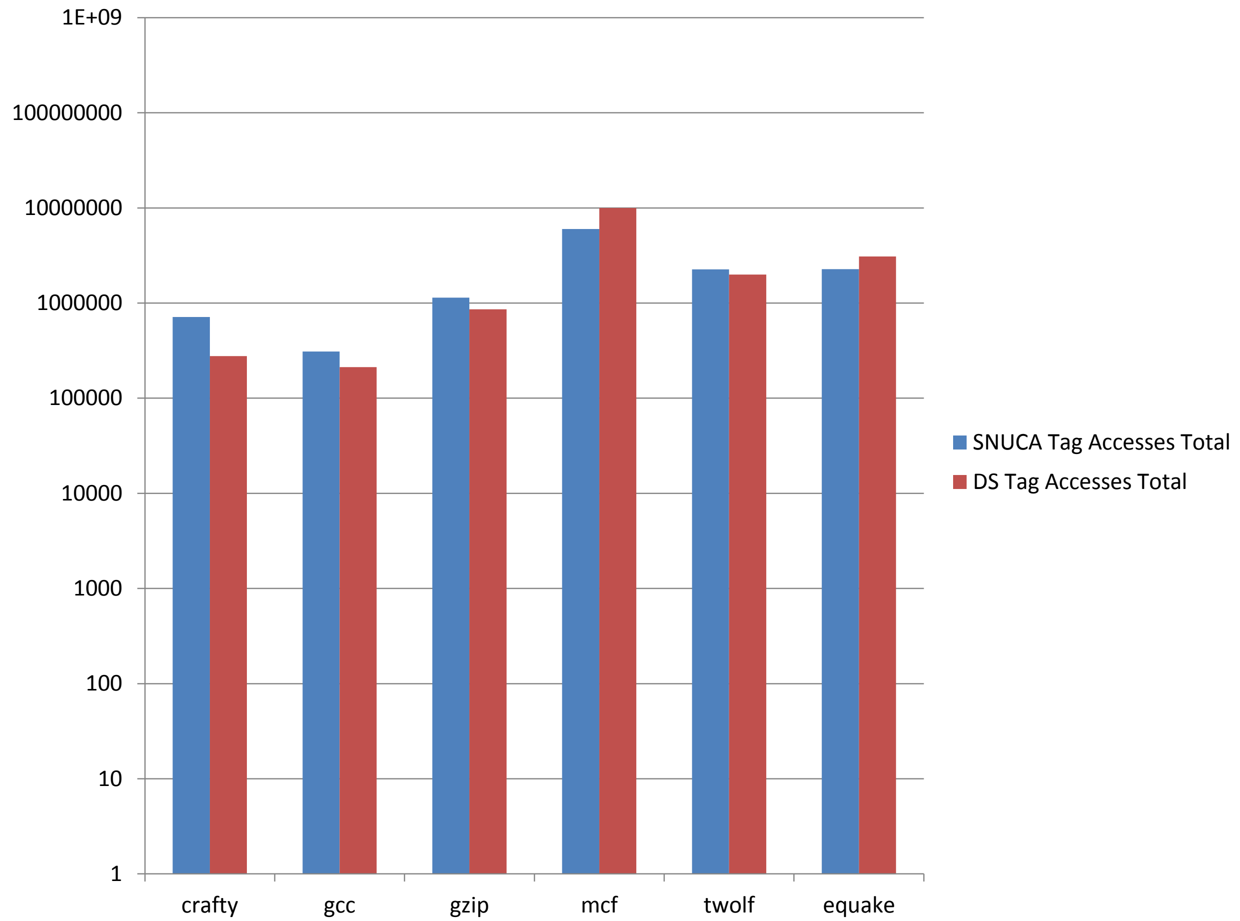

Figure 6.16: The total counts of tag latency accesses. 512 KB bank size, 8 MB in total.

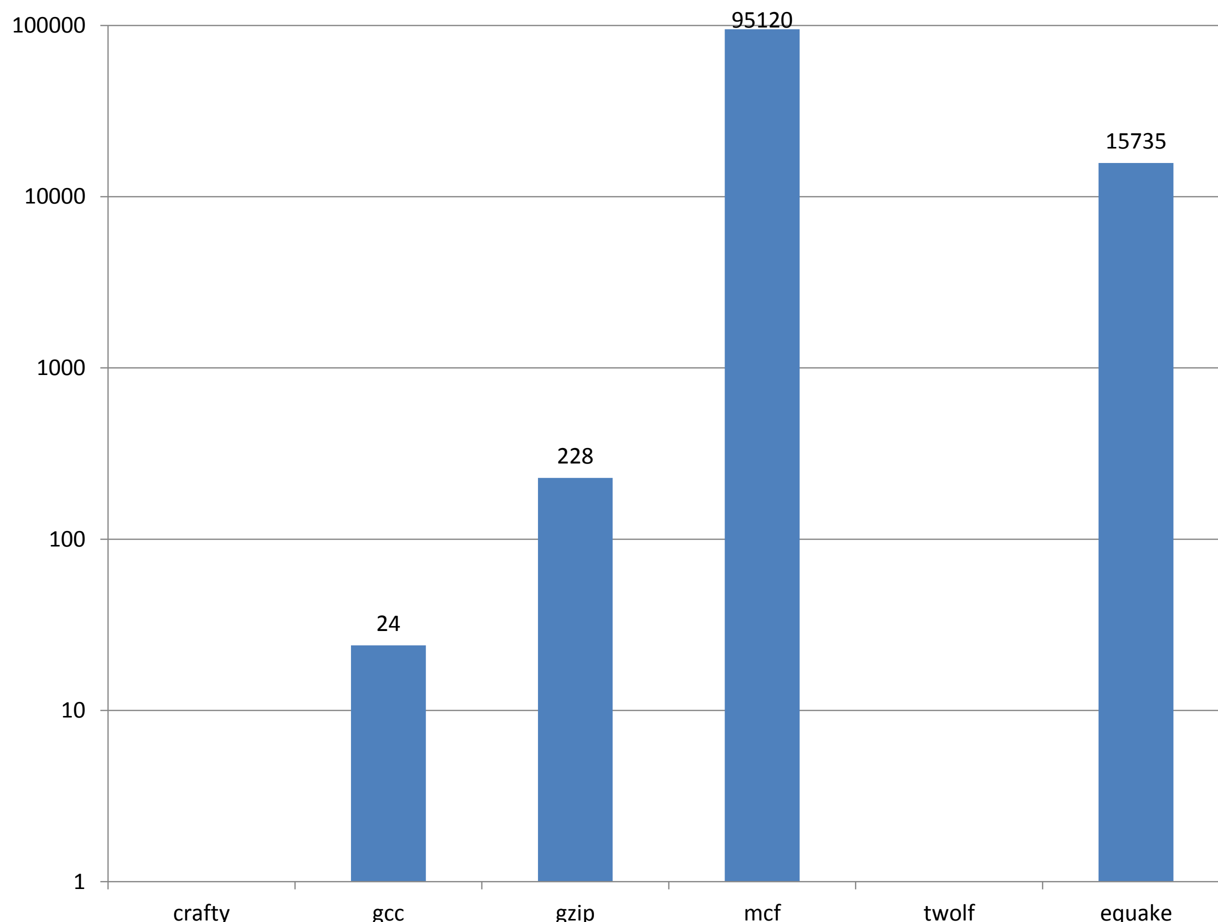

Figure 6.17: The number of page flushes. 512 KB bank size, 8 MB in total.

The bank conflict counts shown in Figure 6.15 reduced significantly from 2 MB bank cases except `mcf` and `equake` result on the DS cache while the tag latency accesses in Figure 6.16 are quite similar with the 2 MB bank case in Figure 6.12. The number of page flushes in Figure 6.17 are slightly reduced over 2 MB case. The maximum, minimum, and average page flush time of 512 KB bank case is depicted in Figure 6.17. The maximum page flush times are consistently smaller than those of 2 MB cases due to decreased access latency while minimum page flush times are quite close because of the same tag access latency. Overall, there are no strong indication why the DS performs better than the SNUCA with increased bank count. This is quite counter-intuitive because the degree of interleaving of the SNUCA cache is increased, so the bank conflict effects more this time.

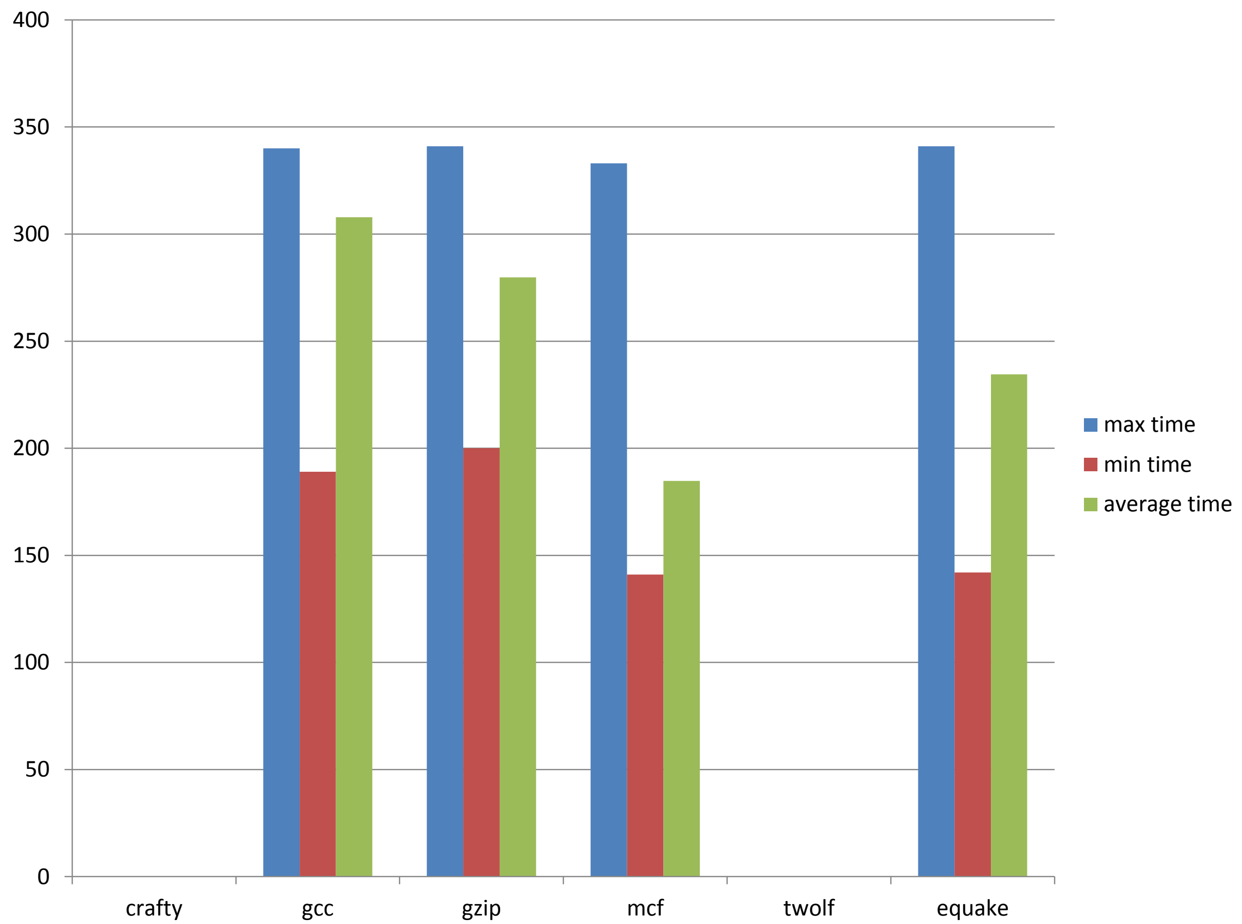

Figure 6.18: Maximum, minimum and average page flush time comparison. 512 KB bank size, 8 MB in total.

One possible explanation is that benefits from the decreased bank conflicts counts are overshadowed by the increased network latency (due to increased average hop counts). Average access time (access latency to a bank + average hop counts) is increased to 8 cycles from 7 cycles. However, it is the same with the DS cache. I examined bank access counts and noticed that because of inefficiency of LRU policy, there are cases of strong bias in the usage of banks in the DS cache while the banks of the SNUCA cache is used quite uniformly. At `crafty`, for example, most of accesses are on the 8th and 9th banks which have hop counts of 3 and 4 while no access on the farthest banks.

Chapter 7

Conclusion

We have discussed design challenges for the future cache design, especially focusing on the last level shared cache. Explicit resource addressing is the design strategy we selected as a solution, and made discussions how it could solve design issues from managing power, performance, and resource sharing efficiency. Also, it would be helpful to integrate the mechanisms onto a single chip.

The Data Shepherding cache is a design to implement the design strategy of explicit resource addressing. On top of that, we designed the cache design more efficient on the large size shared cache. Because of scratch pad bank design, it achieved better area efficiency which lead to have more headroom for scaling in size when we set average access time limit. We called this analysis as average access time analysis. It is done under assumption that accesses to banks are uniformly distributed and there are no page flush/cache replacement. Cache parameters for this analysis is from the CACTI tool. Smaller area is helpful to reduce leakage power, too. 7% of leakage power on average is saved when we compared with SNUCA cache. The comparison is made with leakage power from banks.

Performance analysis results reveals some characteristics of the Data Shepherding which can lead to performance benefits. First characteristic comes from the structural benefit. Even when the access latencies are the same, tag access latency can be smaller because of smaller tag array size. If there are lots of tag latency accesses such as coherence requests on the tag array, the Data Shepherding cache can be performance benefits from the smaller bank conflict windows. Another characteristic is more promising because the performance benefit from it is effective even when both tag and data array access latencies are the same with the SNUCA. If the pages which are to be flushed are filled up with (relatively) many dirty cache lines, page flush can work better than cache replacements. We can consider it as bulk cache replacements.

In conclusion, the Data Shepherding is designed to satisfy future design requests where many design goals are to be fulfilled. Also, it has good features for scaling in size, which would make the Data Shepherding cache a good candidate for the future large scale system design.

Chapter 8

Future Research

We suggested the Data Shepherding cache, and verified some characteristics. Still, there are features untested, especially related to the adaptability of multiple design goals. Also, we did not explore methodologies to fully utilize the verified design benefits in the implementation of goal driven mechanisms. In the following, possible future works are categorized in two ways.

Mechanisms to maximize benefits from the Data Shepherding cache

As the evaluation targets the base line performance of the Data Shepherding cache, we tried to avoid adding mechanisms for performance optimization. For example, LRU methods blindly maps actively working pages onto the same bank which drove performance issues such as increased bank conflicts and more hops to access data. Following work items are possible to improve performance:

Mechanism to avoid bank conflict We can improve in two ways. First method is to increase sub-banking. However, this method adds extra wires which will increase area

of a bank, so the degree of sub-banking is limited by the cost of increased area and access latency. Second method is re-mapping of a page. This method also can be combined with replacement policy to avoid banks which experiences lots of accesses. The second methods needs good decision algorithm to compensate the performance overhead when a page is migrated into a different bank.

Optimization to improve page flush For the baseline performance, there was no distinction between pages to be flushed. However, there are many pages which used only partially, and some pages are read-only such as pages from instruction fetch. Because TLB structure is utilized, it is possible to know the status of a page, so we can utilize it to improve performance of page flush.

Coherence protocol optimization Even though the Data Shepherding cache does not have cache replacement, states and operations for the cache replacement are used for the process of cache line flush. Because of cascading flush from upper layer to the last level shared cache, we can make states and operations simpler to improve performance.

Page replacement policy optimization LRU policy is used as a baseline. As we have seen in the experiment, the policy can impact to the performance badly. We can develop a replacement policy which considers distance from cores and bank conflicts.

Many items describes here actually require more accurate model of TLB. The evaluation of this work used System call Emulation (SE) mode of GEM5, but for the future work, Full System emulation (FS) mode will be needed.

Designs for multiple design goals

The Data Shepherding cache actually has good characteristics even on each design goals. Exploring designs for each design goal of power, performance and fair sharing are among the top priorities for the future work.

Power efficiency The Data Shepherding cache has many good features for the power efficiency. Because the Data Shepherding cache knows overall map of the page placement, we can develop replacement policy to utilize power gating with the granularity of a bank. Access mode of a bank also can be an interesting optimization for the Data Shepherding cache. As sequential access mode on a bank with direct-mapped cache or scratchpad memory is popular to save power [61], we can utilize different access mode on the Data Shepherding design.

Fair resource sharing Set-associativity is not a concern on the Data Shepherding cache, so we can approach fair cache sharing differently with the Data Shepherding cache. There are fair cache sharing designs with bank granularity. One of the notable designs is shown in [21]. Bank granularity sharing is decided with software approach while partitioning inside bank depends on hardware mechanism. The reason for the complex interaction with software is partly due to the multiple granularities which work on the same cache system. The Data Shepherding cache is managed with the granularity of a page, but each bank is explicitly addressed to a bank. So, more hardware dependent in functionalities but also more portable approach is possible due to abstraction can be possible.

Performance Bigger granularity will give access to the new methodology of bulk data transfer. This will be helpful to improve link utilization between the CPU chip and the main memory, but also will be helpful to improve performance as we discussed with the analogy of skyscraper in Chapter 3.

Also, it is another important future task to show that mechanisms from the different domain can be implemented on the same Data Shepherding cache with relatively smaller design efforts to prove efficiency of the Data Shepherding cache.

Bibliography

[1] Benchmarks put iPad Pros A9x chip roughly on par with Intels 2013 Core i5. `http://www.idownloadblog.com/2015/11/12/ipad-pro-geekbench/`.

[2] Elpida begins sample shipments of DDR3 SDRAM (x32) based on TSV stacking technology | news room | elpida memory. `http://www.elpida.com/en/news/2011/06-27.html`.

[3] Elpida memory, inc. `http://www.elpida.com/en/index.html`.

[4] Intel 64 and IA-32 Architectures Software Developer Manuals. `http://intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[5] Intel's Core i7-6700k 'Skylake' processor reviewed. `http://techreport.com/review/28751/intel-core-i7-6700k-skylake-processor-reviewed/2`.

[6] SPEC CPU2000. `https://www.spec.org/cpu2000/`.

[7] SPEC CPU2000 Memory Footprint. `https://www.spec.org/cpu2000/analysis/memory/`.

[8] Xilinx inc. `http://www.xilinx.com/`.

[9] A. Alameldeen and D. Wood. Interactions Between Compression and Prefetching in Chip Multiprocessors. In *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. HPCA 2007*, pages 228–239, Feb. 2007.

[10] T. Austin, V. Bertacco, S. Mahlke, and Y. Cao. Reliable systems on unreliable fabrics. *IEEE Design Test of Computers*, 25(4):322 –332, Aug. 2008.

[11] C. Auth, C. Allen, A. Blattner, D. Bergstrom, M. Brazier, M. Bost, M. Buehler, V. Chikarmane, T. Ghani, T. Glassman, R. Grover, W. Han, D. Hanken, M. Hattendorf, P. Hentges, R. Heussner, J. Hicks, D. Ingerly, P. Jain, S. Jaloviar, R. James, D. Jones, J. Jopling, S. Joshi, C. Kenyon, H. Liu, R. McFadden, B. McIntyre, J. Neirynck, C. Parker, L. Pipes, I. Post, S. Pradhan, M. Prince, S. Ramey, T. Reynolds, J. Roesler, J. Sandford, J. Seiple, P. Smith, C. Thomas, D. Towner, T. Troeger, C. Weber, P. Yashar, K. Zawadzki, and K. Mistry. A 22nm high performance and low-power CMOS technology featuring fully-depleted tri-gate transistors,

self-aligned contacts and high density MIM capacitors. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 131–132, June 2012.

[12] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 2–2. USENIX Association, 2011.

[13] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[14] A. Banerjee, T. Mukherjee, G. Varsamopoulos, and S. K. S. Gupta. Cooling-aware and thermal-aware workload placement for green HPC data centers. In *Green Computing Conference, 2010 International*, pages 245–256, Aug. 2010.

[15] A. Bardine, P. Foglia, G. Gabrielli, C. Prete, and P. Stenström. Improving power efficiency of d-nuca caches. *ACM SIGARCH Computer Architecture News*, 35(4):53–58, 2007.

[16] L. A. Barroso and M. Dubois. The Performance of Cache-coherent Ring-based Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 268–277, New York, NY, USA, 1993. ACM.

[17] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[18] B. Batson and T. Vijaykumar. Reactive-associative caches. In *2001 International Conference on Parallel Architectures and Compilation Techniques, 2001. Proceedings*, pages 49–60, 2001.

[19] J. Bautista. Tera-scale computing and interconnect Challenges3D stacking considerations. *Tutorial, ISCA*, 2008.

[20] B. Beckmann and D. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *37th International Symposium on Microarchitecture, 2004. MICRO-37 2004*, pages 319–330, 2004.

[21] N. Beckmann and D. Sanchez. Jigsaw: Scalable Software-defined Caches. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 213–224, Piscataway, NJ, USA, 2013. IEEE Press.

[22] H. Berghel. The Cost of Having Analog Executives in a Digital World. *Commun. ACM*, 42(11):11–15, 1999.

[23] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 62–63, 2011.

[24] S. Bhunia, S. Mukhopadhyay, and K. Roy. Process variations and process-tolerant design. In *, 20th International Conference on VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems*, pages 699 –704, Jan. 2007.

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[26] I. Bolsens. 2.5-D ICs: just a stepping stone or a long term alternative to 3-D? *SEMICON*, 2012.

[27] B. Bowhill, B. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, C. Morganti, C. Houghton, D. Krueger, O. Franza, J. Desai, J. Crop, D. Bradley, C. Bostak, S. Bhimji, and M. Becker. 4.5 The Xeon ®; processor E5-2600 v3: A 22nm 18-core product family. In *Solid- State Circuits Conference - (ISSCC), 2015 IEEE International*, pages 1–3, Feb. 2015.

[28] W. Bowhill, R. Allmon, S. Bell, E. Cooper, D. Donchin, J. Edmondson, T. Fischer, P. Gronowski, A. Jain, P. Kroesen, B. Loughlin, R. Preston, P. Rubinfeld, M. Smith, S. Thierauf, and G. Wolrich. A 300 MHz 64 b quad-issue CMOS RISC microprocessor. In *Solid-State Circuits Conference, 1995. Digest of Technical Papers. 41st ISSCC, 1995 IEEE International*, pages 182–183, Feb. 1995.

[29] R. Brederlow, W. Weber, D. Schmitt-Landsiedel, and R. Thewes. Hot-carrier degradation of the low-frequency noise in MOS transistors under analog and RF operating conditions. *IEEE Transactions on Electron Devices*, 49(9):1588 – 1596, Sept. 2002.

[30] P. Cappelletti. Non volatile memory evolution and revolution. In *2015 IEEE International Electron Devices Meeting (IEDM)*, pages 10.1.1–10.1.4, Dec. 2015.

[31] L.-P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, page 11261130, New York, NY, USA, 2007. ACM.

[32] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. Opal: a single address space system for 64-bit architecture address space. In *, Third Workshop on Workstation Operating Systems, 1992. Proceedings*, pages 80–85, Apr. 1992.

[33] J. B. Chen, A. Borg, and N. P. Jouppi. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 114–123, New York, NY, USA, 1992. ACM.

[34] L. Cico, R. Cooper, and J. Greene. Performance and programmability of the IBM/Sony/Toshiba Cell broadband engine processor. In *Proceedings of Workshop on Edge Computing Using New Commodity Architectures (EDGE).*

[35] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach.* Gulf Professional Publishing, 1999.

[36] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct. 1974.

[37] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb. 2010.

[38] B. Falsafi, M. Stan, K. Skadron, N. Jayasena, Y. Chen, J. Tao, R. Nair, J. Moreno, N. Muralimanohar, K. Sankaralingam, and C. Estan. Near-Memory Data Services. *IEEE Micro*, 36(1):6–13, Jan. 2016.

[39] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 211–222, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[40] R. Fish. The future of computers - Part 1: Multicore and the Memory Wall | EDN, Nov. 2011.

[41] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 148–157. IEEE, 2002.

[42] P. B. Galvin, G. Gagne, and A. Silberschatz. *Operating system concepts*. John Wiley & Sons, Inc., 2013.

[43] S. Ghosh and K. Roy. Parameter variation tolerance and error resiliency: New design paradigm for the nanoscale era. *Proceedings of the IEEE*, 98(10):1718 –1751, Oct. 2010.

[44] R. Gioiosa. Towards sustainable exascale computing. In *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, pages 270–275. IEEE, Sept. 2010.

[45] L. Gwennap. Sandy Bridge spans generations. *Microprocessor Report (www. MPRonline. com)*, 2010.

[46] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2):6–20, 2014.

[47] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 184–195, New York, NY, USA, 2009. ACM.

[48] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6 –15, Aug. 2011.

[49] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[50] J. Hu, C. Xue, Q. Zhuge, W.-C. Tseng, and E.-M. Sha. Towards energy efficient hybrid on-chip Scratch Pad Memory with non-volatile memory. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1 –6, Mar. 2011.

[51] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 31–40, New York, NY, USA, 2005. ACM.

[52] ITRS. 2009 Edition of ITRS Edition Reports and Ordering. 2009.

[53] T. Jain and T. Agrawal. The Haswell microarchitecture4th generation processor. *International Journal of Computer Science and Information Technologies*, 4(3):477–480, 2013.

[54] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589 –604, July 2005.

[55] D. Kanter. Intel's Sandy Bridge Microarchitecture. `http://www.realworldtech.com/sandy-bridge/7/`.

[56] D. Kaseridis, J. Stuecheli, and L. John. Bank-aware Dynamic Cache Partitioning for Multicore Architectures. In *International Conference on Parallel Processing, 2009. ICPP '09*, pages 18–25, Sept. 2009.

[57] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. *ACM SIGARCH Computer Architecture News*, 29(2):240–251, 2001.

[58] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Acm Sigplan Notices*, volume 37, pages 211–222. ACM, 2002.

[59] I. Kotera, K. Abe, R. Egawa, H. Takizawa, and H. Kobayashi. Power-aware dynamic cache partitioning for cmps. *Transactions on High-Performance Embedded Architectures and Compilers*, 3(2):149–167, 2008.

[60] M. E. Langer. Intel corporation - Intel reinvents transistors using new 3-D structure. `http://www.intc.com/releasedetail.cfm?ReleaseID=574448&ReleasesType=&wapkw=tri-gate+transistors`.

[61] T. Lanier. Exploring the design of the cortex-a15 processor. `http://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf`, 2011.

[62] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143, Feb. 2010.

[63] X. Li and J.-L. Gaudiot. Tolerating radiation-induced transient faults in modern processors. *International Journal of Parallel Programming*, 38(2):85–116, Apr. 2010.

[64] X. Liang, G.-Y. Wei, and D. Brooks. ReVIVaL: a variation-tolerant architecture using voltage interpolation and variable latency. In *35th International Symposium on Computer Architecture, 2008. ISCA '08*, pages 191 –202, June 2008.

[65] R. Maddah, S. Cho, and R. Melhem. Data dependent sparing to manage better-than-bad blocks. *Computer Architecture Letters*, PP(99):1, 2012.

[66] M. R. Marty and M. D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006. MICRO-39*, pages 309–320, Dec. 2006.

[67] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[68] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: The Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[69] G. Memik, G. Reinman, and W. Mangione-Smith. Just say no: benefits of early cache miss determination. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*, pages 307–316, Feb. 2003.

[70] J. F. Meyer. Performability: a retrospective and some pointers to the future. *Performance Evaluation*, 14(34):139–156, Feb. 1992.

[71] K. Mistry, C. Allen, C. Auth, B. Beattie, D. Bergstrom, M. Bost, M. Brazier, M. Buehler, A. Cappellani, R. Chau, C. H. Choi, G. Ding, K. Fischer, T. Ghani, R. Grover, W. Han, D. Hanken, M. Hattendorf, J. He, J. Hicks, R. Huessner, D. Ingerly, P. Jain, R. James, L. Jong, S. Joshi, C. Kenyon, K. Kuhn, K. Lee, H. Liu, J. Maiz, B. McIntyre, P. Moon, J. Neirynck, S. Pae, C. Parker, D. Parsons, C. Prasad, L. Pipes, M. Prince, P. Ranade, T. Reynolds, J. Sandford, L. Shifren, J. Sebastian, J. Seiple, D. Simon, S. Sivakumar, P. Smith, C. Thomas, T. Troeger, P. Vandervoorn, S. Williams, and K. Zawadzki. A 45nm Logic Technology with High-k+Metal Gate Transistors, Strained Silicon, 9 Cu Interconnect Layers, 193nm Dry Patterning, and 100% Pb-free Packaging. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 247–250, Dec. 2007.

[72] N. Mohyuddin, R. Bhatti, and M. Dubois. Controlling leakage power with the replacement policy in slumberous caches. In *Proceedings of the 2nd conference on Computing frontiers*, pages 161–170. ACM, 2005.

[73] D. Molka, D. Hackenberg, R. Schne, and W. E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *2015 44th International Conference on Parallel Processing (ICPP)*, pages 739–748, Sept. 2015.

[74] Y. Morita, H. Fujiwara, H. Noguchi, Y. Iguchi, K. Nii, H. Kawaguchi, and M. Yoshimoto. An Area-Conscious Low-Voltage-Oriented 8t-SRAM Design under DVS Environment. In *2007 IEEE Symposium on VLSI Circuits*, pages 256–257, June 2007.

[75] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 234–245, Washington, DC, USA, 2005. IEEE Computer Society.

[76] S. Mukhopadhyay, K. Kim, H. Mahmoodi, and K. Roy. Design of a process variation tolerant self-repairing SRAM for yield enhancement in nanoscaled CMOS. *IEEE Journal of Solid-State Circuits*, 42(6):1370 –1382, June 2007.

[77] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007. MICRO 2007*, pages 3–14, Dec.

[78] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[79] W. Najjar and J.-L. Gaudiot. Scalability analysis in gracefully-degradable large systems. *IEEE Transactions on Reliability*, 40(2):189 –197, June 1991.

[80] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[81] T. Nigam, A. Kerber, and P. Peumans. Accurate model for time-dependent dielectric breakdown of high-k metal gate stacks. In *Reliability Physics Symposium, 2009 IEEE International*, pages 523 –530, Apr. 2009.

[82] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of the 1997 European Conference on Design and Test*, EDTC '97, pages 7–, Washington, DC, USA, 1997. IEEE Computer Society.

[83] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing Set-associative Cache Energy via Way-prediction and Selective Direct-mapping. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 54–65, Washington, DC, USA, 2001. IEEE Computer Society.

[84] M. Qureshi and Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006. MICRO-39*, pages 423–432, Dec. 2006.

[85] G. Reinman and N. P. Jouppi. CACTI 2.0: An integrated cache timing and power model. *Western Research Lab Research Report*, 7, 2000.

[86] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM.

[87] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. M. L. M. Palma, M. Dayd, O. Marques, and J. C. Lopes, editors, *High Performance Computing for Computational Science VECPAR 2010*, volume 6449, pages 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[88] J. W. Sheaffer, D. P. Luebke, and K. Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, page 5564, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[89] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[90] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *29th Annual International Symposium on Computer Architecture, 2002. Proceedings*, pages 123 – 134, 2002.

[91] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):92, Nov. 2011.

[92] P. Stanley-Marbell, V. Caparros Cabezas, and R. Luijten. Pinned to the Walls: Impact of Packaging and Application Properties on the Memory and Power Walls. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pages 51–56, Piscataway, NJ, USA, 2011. IEEE Press.

[93] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 72–82, New York, NY, USA, 2010. ACM.

[94] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[95] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen. A novel architecture of the 3D stacked MRAM l2 cache for CMPs. In *IEEE 15th International Symposium on High Performance Computer Architecture, 2009. HPCA 2009*, pages 239 –249, Feb. 2009.

[96] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical report, Technical Report HPL-2006-86, HP Laboratories Palo Alto, 2006.

[97] M. Taylor. A Landscape of the New Dark Silicon Design Regime. *IEEE Micro*, 33(5):8–19, Sept. 2013.

[98] S. E. Thompson and S. Parthasarathy. Moore's law: the future of Si microelectronics. *Materials Today*, 9(6):20–25, June 2006.

[99] M. Tolentino and K. W. Cameron. The Optimist, the Pessimist, and the Global Race to Exascale in 20 Megawatts. *Computer*, 45(1):95–97, Jan. 2012.

[100] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

[101] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. *ACM Comput. Surv.*, 37(3):195–237, Sept. 2005.

[102] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 340–349, Oct. 2011.

[103] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim. Workload and Power Budget Partitioning for Single-chip Heterogeneous Processors. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 401–410, New York, NY, USA, 2012. ACM.

[104] G. I. Wirth, M. G. Vieira, E. H. Neto, and F. L. Kastensmidt. Modeling the sensitivity of CMOS circuits to radiation induced single event transients. *Microelectronics Reliability*, 48(1):29–36, Jan. 2008.

[105] D. H. Woo, N. H. Seong, D. Lewis, and H.-H. Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Jan. 2010.

[106] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM.