ABSTRACT

GUPTA, SAURABH. Locality Driven Memory Hierarchy Optimizations. (Under the direction of Huiyang Zhou.)

This dissertation first revisits the fundamental concept of the locality of references and proposes to quantify locality as a conditional probability: in an address stream, given the condition that an address is accessed, how likely the same address (temporal locality) or an address within its neighborhood (spatial locality) will be accessed in the near future. Based on this definition, spatial locality is a function of two parameters, the neighborhood size and the scope of near future, and temporal locality becomes a special case of spatial locality with the neighborhood size being zero byte. In previous research, some ad-hoc metrics have been proposed as a quantitative measure of spatial locality. In contrast, conditional probability-based locality measure has a clear mathematical meaning, offers justification for distance histograms, and provides a theoretically sound and unified way to quantify both temporal and spatial locality. The proposed locality measure clearly exhibits the inherent application characteristics, from which it is easy to derive information such as the sizes of the working data sets and how locality can be exploited. We showcase that our quantified locality visualized in 3D-meshes can be used to evaluate compiler optimizations, to analyze the locality at different levels of memory hierarchy, to optimize the cache architecture to effectively leverage the locality, and to examine the effect of data prefetching mechanisms.

Secondly, we present an online locality monitoring framework to capture locality information at runtime. We observe that for many memory-intensive workloads, their cache capacity requirements vary significantly for different cache-line sizes. In single-core processors, choosing the right cache-line sizes improves the performance and enables a significant portion of a cache to be turned off when there is strong spatial locality. For multi-core processors, we use our online locality monitoring to drive cache partitioning. We highlight that exploiting spatial locality enables much more effective cache sharing by reducing capacity for workloads with strong spatial locality and making them effective
donors for other co-scheduled workloads. We leverage our online locality monitoring to determine both the proper block size and the capacity assigned to each workload in a shared LLC. Our experiments show that our proposed approaches deliver significant energy savings in both single-core and multi-core processors.

Third, we propose a microarchitecture solution to enable cache bypassing for inclusive cache hierarchy designs. Recent works shows that cache bypassing is an effective technique to enhance the last level cache (LLC) performance. However, commonly used inclusive cache hierarchy cannot benefit from this technique because bypassing inherently breaks the inclusion property. We propose that bypassed cache lines can skip the LLC while their tags are stored in a ‘bypass buffer’. Our key insight is that the lifetime of a bypassed line, assuming a well-designed bypassing algorithm, should be short in upper level caches and is most likely dead when its tag is evicted from the bypass buffer. Therefore, a small bypass buffer is sufficient to maintain the inclusion property and to reap most performance benefits of bypassing. We show that a top performing cache bypassing algorithm, which is originally designed for non-inclusive caches, performs comparably for inclusive caches equipped with our bypass buffer. Furthermore, the bypass buffer facilitates bypassing algorithms by providing the usage information of bypassed lines to which reduces hardware implementation cost significantly compared to the original design.

In summary, this dissertation proposes new quantitative measure of locality of references; derives insights to find opportunities of memory hierarchy optimizations, and presents online locality monitoring scheme to enable runtime cache optimizations for energy efficiency and high performance.
Locality Driven Memory Hierarchy Optimizations

by

Saurabh Gupta

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2014

APPROVED BY:

__________________________________________  ______________________________
Eric Roternberg                               Xiaosong Ma

__________________________________________  ______________________________
Gregory Byrd                                   Huiyang Zhou
   Chair of Advisory Committee
DEDICATION

To my parents, brothers, grandparents, family, friends and teachers.
BIOGRAPHY

Saurabh Gupta was born to Shri Hari Om Gupta and Srimati Krishna Gupta in Kanpur, an industrial city in the state of Uttar Pradesh in India. He obtained his primary education in the city of Kanpur at schools including Pt. DeenDayal Upadhyay Sanatan Dharm Vidhyalaya. He has obtained his Bachelors and Masters in Electrical Engineering from Indian Institute of Technology Kanpur in 2009. Thereafter he joined Computer Engineering PhD program at NC State University in fall 2009.

He has been working with Dr. Huiyang Zhou for past 4 years and has worked on several research projects related to computer architecture research focusing on novel cache architectures and memory hierarchy optimization. One of his publications won the best paper award in architecture track at IEEE IPDPS 2012. He has done summer internships with ORNL, Intel and IBM while he was a student. Besides research, he is interested in teaching the fundamentals of computer system design. He is a member of IEEE, ACM and IEEE Computer Society.
ACKNOWLEDGEMENTS

First and foremost I would like to acknowledge the support and encouragement of my parents and family to pursue my career goals and higher studies. Now, I would like to take this opportunity to acknowledge a lot of people who have contributed to my journey of finishing the formal education (loosely keeping them in a chronological order),

A job well begun is half done. I was lucky to start my education in the guidance of my father, my mother and my tutor, Anuj Sir. During high school, I admired the teaching style of Mahesh ji and Kailash ji who helped me attain good mathematical skills. During my year of break from school, Anish Sir and Pankaj Sir honed my command on physics and chemistry respectively to the competitive level. Also, I am grateful to my brothers/cousins, Sachin Gupta, Nirmit Gupta, Ramakant Gupta, Gaurav Gupta and Pradeep Gupta for being my best friends throughout my childhood. My friends from high school also had a great influence on shaping me and my career. I am especially thankful to Mayank Shukla, Suyash Mishra, Anunay Gupta, Srot Gupta, Ashish Kushwaha, Aditya Tiwari, Saurabh Awasthi, Abhinav Tripathi, Anurag Verma, Anurag Sharma, Pranjal Srivastava, Sugam Srivastava, Lokesh Dwivedi, Manoj Tripathi for being great friends. I am also thankful to Mayank Shukla, Suyash Mishra, Anunay Gupta, Srot Gupta, Ashish Kushwaha, Aditya Tiwari, Saurabh Awasthi, Abhinav Tripathi, Anurag Verma, Anurag Sharma, Pranjal Srivastava, Sugam Srivastava, Lokesh Dwivedi, Manoj Tripathi for being great friends. I am also thankful to Mayank Shukla, Suyash Mishra, Anunay Gupta, Srot Gupta, Ashish Kushwaha, Aditya Tiwari, Saurabh Awasthi, Abhinav Tripathi, Anurag Verma, Anurag Sharma, Pranjal Srivastava, Sugam Srivastava, Lokesh Dwivedi, Manoj Tripathi for being great friends. I am also thankful to Gaurav Gupta, Arun Koshta, Vikalp Sachan and Vikhyat Umrao for being there while we prepared for the toughest exam I ever wrote (at least it seemed so then). This led me to IIT Kanpur for my undergrad.

IIT Kanpur turned out to be everything I had hoped for and more. In my undergraduate studies, I was a fan of Dr. Amitabha Ghosh, Dr. I. D. Dhariyal, Dr. Brij Bhusan and Dr. Mainak Chaudhuri. I am thankful to Veeramani V. and Abhinav Agarwal, who introduced me to the area of Computer Architecture in EE-summer camp 2006 and since then I never stopped appreciating it as an art. I am also grateful to Peter Hofstee and Manish Gupta for recommending me for summer internship at IBM, and to Shakti Kapoor, Shubhodeep Roy Choudhury and Prasenjit Chakraborty for making it a successful and great learning experience. I would especially like to thank Dr. Mainak Chaudhuri for taking me up as his M.
Tech. thesis student and Dr. S. S. K. Iyer in supporting my interdisciplinary thesis venture. I am also thankful to both of them for encouraging me to pursue a PhD program after my master degree. I would also like to thank Dr. S. P. Das and Dr. Rajat Moona for serving on my M. Tech. thesis committee at IIT Kanpur. Also, I am thankful to Dr. Y. N Singh and Dr. S. Qureshi for providing their recommendations with my application to PhD program.

Apart from academics at IIT Kanpur, my wing-mates including Purushottam Kar, Nikhil Saurabh, Vibhu Singh, Mayank Agarwal, Ravi Gupta, Vikas Sharma, Vishal Sharma, Suhail Rizvi, Prashant Saxena, Rajat Goyal, Soumya Banik, Pratibh Agarwal, Vivek Alok, Ankit Gupta, Samanvay Srivastava, Cherian Matthew, Shyam Sundar, Nipun Jain, Hemant Kumar, Vijay Bharti, Abhishek Sharma, Mayank Pandey, Vikas Saxena, Rohit Bishnoi, Vineet Dwivedi and Shreyansh Jain were the most motivating and supporting community I could have ever hoped for (B-mid ka tempo high hai). I would also like to acknowledge fellow EE student at IIT Kanpur including Ajit Pratap Singh, Mukund, Krishna Teja, K Ashwin, and Ashish Dembla. I had great time working with Deepak Ailani on my first ever research project and I was extremely blessed to have Nitin Munjal as my project partner for my first ever computer architecture course. I am especially thankful to Abhayendra Narayan Singh for helping me throughout my M. Tech. thesis with variety of things including application process for PhD program. At NC State, I survived my first semester with the guidance of Dr. Eric Rotenberg, Amit Qusba, Devesh Tiwari and Siddharth Chabbra. I also would like to thank Dr. Huiyang Zhou and Dr. Yan Solihin for very patiently talking to me at many occasions while I was trying to find a research focus to pursue for this dissertation.

Working with Dr. Huiyang Zhou, I have learned great research ethic in past 4 years. His result and evidence oriented style is something that I will treasure for long. I am especially grateful to him for his patience with me, and the time he devoted discussing my research in great depth and providing valuable feedback. Without his support and guidance this dissertation would not have been possible. I am also thankful to him for letting me explore my interest in teaching by letting me cover a few lectures for his class.
I would also like to thank Dr. Eric Rotenberg, Dr. Greg Byrd and Dr. Xiaosong Ma for serving on my dissertation committee and providing their critique as well as encouragement on my research projects. I have been a fan of the teaching style of Dr. Rotenberg and Dr. Byrd, and I want to be appreciated as much by my students one day.

I had brilliant colleagues at NCSU. I would like to thank Nitin Kwatra, Pawandeep Singh Taluja, Sandeep Navada, Siddhartha Chabbra, Brandon Dwiel, Niket Chaudhary, Elliot Forbes, Ahmad Samih, Rami Al Sheikh, Amro Awad, Hiran Mayukh, Ganesh Balakrishnan, Yi Yang, Ping Xiang, Qi Jha, Xiangyang Guo, Yuan Liu, Hongwen Dai and Chao Li for being helpful during the course of my PhD program.

As they say, you are only as good as your company you keep. I am grateful to Devesh Tiwari, Rajeshwar Vanka, Amit Qusba, Avik Juneja, Raj Kumar, Rahul Ramasubramanian, Mahesh BV, Michelle Schisa, Aditya Deorha, Arpit Gupta, Santosh Navada, Sahil Sabharwal, Rohit Taneja and Ajit Narwal for being there at a more personal level of friendship throughout these years. I must acknowledge Devesh’s ability to find related literature that always amazes me and has helped me on quite a few occasions. I would also like to thank Avik Juneja, Mahesh BV, Aditya Deorha, Santosh Navada, Ravi Komanduri, Kok Ren, Sriram PS and Neeraj Gole for being available for sports to keep my sanity. Special thanks to Arup Mazumdar, Dayanand Shaldhul and Mohit Gupta for tagging along with me on numerous road trips.

I am thankful to Linda Fontes, Elaine Hardin, Jennifer Raab and Kendall Del Rio for providing outstanding administrative support throughout the years. I cannot thank them enough for what they do for us as graduate students.

Last but certainly not the least; I would like to thank Dr. Hongliang Gao for collaborating with me on one of my research projects. I would also like to thank National Science Foundation (CNS-0905223 and CCF-0968667) and Intel for supporting my research in part through research grants. Any opinions, findings, and conclusions expressed in this dissertation are those of the authors and do not necessarily reflect the views of the National Science Foundation.
# TABLE OF CONTENTS

LIST OF TABLES----------------------------------------------------------------------------------------------------------------- x
LIST OF FIGURES ----------------------------------------------------------------------------------------------------------------- xi

Chapter 1-------------------------------------------------------------------------------------------------------------------------- 1
1. Introduction and Motivation ------------------------------------------------- 1

Chapter 2-------------------------------------------------------------------------------------------------------------------------- 4
2. Locality Principle Revisited: A Probability-Based Quantitative Approach ------- 4
   2.1. Introduction------------------------------------------------------------- 4
   2.2. Quantifying Locality of Reference Using Conditional Probabilities------- 6
       2.2.1. Temporal Locality-------------------------------------------------- 6
       2.2.2. Spatial Locality---------------------------------------------------- 9
       2.2.3. The Relationship between Spatial and Temporal Locality----------- 13
       2.2.4. Difference Functions of Locality Measures and the Relationship to Reuse Distance Histograms---------- 14
       2.2.5. Sub-trace Locality-------------------------------------------------- 16
   2.3. Locality Analysis for Code Optimizations ------------------------------- 18
   2.4. Locality Analysis for Memory Hierarchy Optimizations ----------------- 20
       2.4.1. Experimental Methodology------------------------------------------- 20
       2.4.2. Locality at Different Memory Hierarchy Levels--------------------- 21
       2.4.3. Memory Hierarchy Optimizations------------------------------------- 22
       2.4.4. Locality Improvement from Data Prefetching------------------------ 29
       2.4.5. Understanding the replacement policy-------------------------------- 33
       2.4.6. Spatial Locality vs. Temporal Locality in Last Level Caches------- 34
   2.5. A GPU-based Parallel Algorithm for Locality Computation----------------- 39
   2.6. Related Work------------------------------------------------------------- 40
   2.7. Summary-------------------------------------------------------------------- 41
Chapter 3
3. Locality Driven Cache Hierarchy Optimizations for Energy Efficient and High Performance Caching
3.1. Introduction
3.2. Related Work
3.4. Locality Driven Cache Management
3.4.1. Dynamically Adjusting Cache Block Sizes
3.4.2. Locality-Driven Cache Reconfiguration for Single-Core Processors: Hcache Algorithm
3.4.3. Spatial Locality-aware Cache Partitioning for Shared LLC in Multicores
3.5. Experimental Methodology
3.6. Experimental Results
3.6.2. Evaluating SLCP for Shared LLCs in Multi-core Processors
3.7. Summary

Chapter 4
4. Adaptive Cache Bypassing for Inclusive Last Level Caches
4.1. Introduction
4.2. Motivation
4.3. Adaptive Cache Bypassing for Inclusive LLCs
4.3.1. Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing (DSB)
4.3.2. Bypass-Buffer Enabled Inclusive DSB
4.3.3. Data-less Bypass Buffers
4.3.4. Efficient Tracking Using Bypass Buffers
LIST OF TABLES

Table 3.1. Performance comparison among different cache configurations .................. 45
Table 3.2. A case study trace of the benchmark gcc............................................... 46
Table 3.3. Cache parameters for 4MB, 16-way set associative LLC .......................... 52
LIST OF FIGURES

Figure 2.1. Calculating temporal locality. (a) Reference trace (also showing a moving future window of size 2); (b) Joint probability using Def. 1 of near future; (c) Temporal locality $TL(N)$ using different definitions of near future...................................................... 8

Figure 2.2. The spatial locality plots of the benchmarks, $mcf$ (SPEC2006) and $bzip2$ (SPEC2000). Temporal locality is a special case of spatial locality with neighborhood size as 0................................................................. 11

Figure 2.3. Spatial locality of the benchmark $sphinx$ for neighborhood definitions. (a) definition 1 (b) definition 2 ................................................................. 12

Figure 2.4. The Venn diagram showing the relationship among events $X_0 = A_1$, $X_1 = A_1 + \Delta$, and $X_2 = A_1$ ................................................................. 13

Figure 2.5. Locality and the difference functions for the benchmark $hmmer$. (a) Spatial Locality (b) Difference function over the near future window size $N$, which is the same as reuse distance histograms (c) Difference function over neighborhood size $K$, which shows a histogram of stride access patterns ...................................................... 15

Figure 2.6. Locality for different matrix multiplication kernels (a) Untiled (b) 32x32 tile (c) 64x16 tile................................................................. 19

Figure 2.7. The spatial locality of the benchmarks, $art$, $mcf$ (SPEC 2000), and $bzip2$ (SPEC2000) at different cache levels...................................................... 21

Figure 2.8. The spatial locality of L2 access traces for various benchmarks............. 23

Figure 2.9. The spatial locality for $wupwise$ (a) L2 access trace (b) L2 miss stream for 1MB L2 cache with a 64B block size (c) L2 miss stream for 1MB L2 cache with a 128B block size (d) L2 miss stream with a stream buffer ...................................................... 25

Figure 2.10. Speedups of larger block sizes, next-n-line prefetching and stream-buffer................................................................................... 26

Figure 2.11. The spatial locality of L2 miss stream for the benchmarks (a) $equake$; (b) $lbm$; (c) $mcf$ (SPEC2000); and (d) $milc$ ........................................... 27

Figure 2.12. Performance improvements from the proposed buffers at the memory controller level................................................................................... 28

Figure 2.13. The locality improvement of $mcf$ and $soplex$ at the L2 cache level (a) the locality of the L2 demand accesses of $mcf$ without the prefetcher, (b) the sub-trace locality of the L2 demand trace of $mcf$ with the prefetcher, (c) the locality of the L2 demand accesses of $soplex$
without the prefetcher, (d) the sub-trace locality of the L2 demand
trace of soplex with the prefetcher.............................................................. 30
Figure 2.14. The locality improvement at L2 cache from data prefetcher for milc,
hummer, lbm and calculix shown by comparing the locality of demand
access trace and sub-trace locality of demand accesses with
prefetcher ............................................................................................................. 31
Figure 2.15. Locality of the LLC access stream for benchmarks (a) xalancbmk,
and (b) milc .................................................................................................... 35
Figure 2.16. IPC speedup of LRU, 3P-4P and DRRIP with different cache block
sizes over the baseline LLC with a 64-byte block size and the LRU
replacement policy ............................................................................................ 36
Figure 2.17. Different types of cache misses for LLCs with the block size of 64
bytes vs. 128 bytes ........................................................................................... 37
Figure 2.18. Weighted speedups of LRU, 3P-4P and DRRIP with different cache
block sizes (4MB LLC) over the baseline 4MB LLC with a 64-byte
block size and the LRU replacement policy ...................................................... 38
Figure 3.1: The locality plot of the benchmark gcc at L3 level ......................... 44
Figure 3.2: Organization of (a) ATDs and Locality Scores (LScores) for locality
computation (b) 2D-array of LScores per core .................................................. 51
Figure 3.3: SLCP Architecture for LLCs ............................................................ 55
Figure 3.4. Normalized (a) L3 energy consumption, IPC Speedup and System
EDP for the Hcache algorithm and the Capacity_only scheme, and
(b) L3 static/dynamic/total energy consumption, DRAM energy
consumption, total energy consumption of the system for Hcache
algorithm ........................................................................................................... 61
Figure 3.5. Normalized weighted speedup of UCP, UCP with static
128B/256B/512B cache block size and ALS ..................................................... 63
Figure 3.6. Normalized IPC throughput and weighted speedup of UCP, ALS,
PriSM and SLCP for 4-way multiprogrammed workloads ............................... 64
Figure 3.7. Weighted speedup of UCP, ALS, PriSM and SLCP relative to the
baseline for 8-way multiprogrammed workloads and geometric-
mean summary for each category .................................................................... 66
Figure 3.8. (a) Normalized IPC throughput for UCP, ALS, UCP+ALS, and
SLCP over the baseline, (b) Normalized weighted speedup of UCP,
ALS, UCP+ALS and SLCP ............................................................................. 67
Figure 3.9. Different hit rate curves used by UCP, PriSM and SLCP while
allocating ways to benchmark mcf ................................................................... 69
Figure 3.1. Variation of average WS improvement of SLCP with different values of β parameter .......................................................... 70
Figure 3.11. Relative Power and Energy consumption of SLCP with respect to the baseline ............................................................................ 71
Figure 3.12. Normalized IPC throughput of UCP, ALS, UCP+ALS and SLCP at L3 cache when ALS is applied at L2 cache ........................................ 72
Figure 4.1. Memory hierarchy organization for (a) a non-inclusive LLC (b) an inclusive LLC (c) a non-inclusive LLC with bypass (selective fill of L3 cache) (d) an inclusive LLC with a bypass buffer to support cache bypassing ...................................................................... 77
Figure 4.2. The lifetime histogram of the blocks, which are bypassed from the LLC, in the L1 data cache .......................................................................... 80
Figure 4.3. Various fields present in a BB-entry .......................................................... 84
Figure 4.4. LLC miss rate comparison for different designs ................................................ 87
Figure 4.5. Fraction of bypassed LLC allocations for I-DSB-BBtracking .............. 88
Figure 4.6. Fraction of bypassed blocks incurring a hit in the bypass buffer for I-DSB-BBtracking ........................................................................ 88
Figure 4.7. Performance improvements of DSB, I-DSB-BB and I-DSB-BBtracking (w.r.t. the baseline inclusive LLC with the LRU replacement policy) ........................................................................ 89
Figure 4.8. Performance of I-DSB-BB and I-DSB-BBtracking for different bypass buffer sizes ........................................................................ 90
Figure 4.9. Performance improvements of DRRIP and I-DSB-BBtracking (w.r.t. the baseline inclusive LLC with the LRU replacement policy) .......... 92
Figure 4.10. Performance improvements of DRRIP and I-DSB-BBtracking in presence of a stream buffer. The baseline is an inclusive LLC with the LRU replacement and a stream buffer prefetcher ........................................................................ 93
Figure 4.11. Performance improvements of DSB, I-DSB-BB and I-DSB-BBtracking for different cache configurations. (the baselines are inclusive LLCs with the corresponding configurations) .................................. 94
Figure 4.12. Performance improvements of I-DSB-BBtracking and TA-DRRIP for a 4MB inclusive LLC for multi-programmed workloads (w.r.t. the baseline with the LRU replacement policy) ........................................... 95
Figure 4.13. Comparing energy consumption of I-DSB-BBtracking with the baseline ..................................................................................... 96
Figure 4.14. Comparing memory bandwidth of I-DSB-BBtracking with the baseline system ..................................................................................... 97
Figure A.1. Locality plots for L3 access traces for SPEC2006 benchmarks ............... 111
Figure A.2. Locality plots for L3 access traces for SPEC2006 benchmarks ............... 112
Figure B.1. Sub-trace locality of L2 demand access traces for SPEC2006 benchmark.................................................................................................................. 114
Figure B.2. Locality of L2 demand accesses and L2 miss traces in absence of prefetching ............................................................................................................. 115
Figure B.3. Locality of L2 demand accesses and L2 miss traces in absence of prefetching ............................................................................................................. 116
Chapter 1

Introduction and Motivation

Memory hierarchy design is known to have significant impact on overall system performance. Modern microprocessor chips (e.g. Intel core i7 processor) have a significant area devoted to on-chip caches [83] which helps in hiding the long DRAM access latency of memory requests. Different levels in the cache hierarchy have different design trade-offs and may prefer different approaches to obtain good performance. Therefore, understanding the locality behavior of programs at different levels of caching is critical for optimizing the overall cache hierarchy. In this research, we first target at defining the spatial and temporal locality under a common mathematical framework. Quantitative measures of temporal locality [48][76] have been studied extensively in the part research but they lack the capability to quantify spatial locality under the same framework. Previous attempts on quantifying spatial locality [4][6][24][68] have taken ad-hoc approaches to suit the end goal of optimization in consideration. On the contrary our goal in this research is to quantify locality using a unified locality measure that is fundamentally simple and has a clear meaning to it. We propose conditional probability based locality measure and showcase that we can obtain many interesting insights from analyzing the locality with the proposed framework discussed in the following chapter. A key insight obtained from this study of locality is that there is a significant spatial locality at lower levels of cache for many applications, which can
be exploited in various ways to improve performance or save energy consumption. This motivates the second chapter of this dissertation where we propose hardware mechanism to estimate spatial locality in an online fashion.

We present detailed architecture of online locality monitors to drive locality driven cache optimizations. Due to large area dedicated to last level caches, significant portion of static energy consumption is contributed by last level cache structure. Previous works focus on lowering cache consumption at little to no cost of performance degradation, such as [1]. Based on our spatial locality observation, we propose a novel cache reconfiguration algorithm to reduce cache energy consumption while improving cache performance. We also propose a novel approach to spatial locality aware cache partitioning for high performance. The cache partitioning has been proposed in the past to provide isolated capacity to each benchmark because uncontrolled management can lead to degraded performance of co-scheduled applications in a multicore system [52]. This problem has been approached from the temporal locality point-of-view in previous works by focusing on capacity management [12][52][56][61][65]. We make a case for using our proposed locality measure to analyze the spatial locality and temporal locality in runtime to drive spatial locality aware cache partitioning. In this manner, our proposed spatial-locality aware cache partitioning creates cache partitions which are heterogeneous in terms of cache block sizing as well as capacity. We show that this leads to significant improvement in performance and energy savings.

Thereafter, we study inclusion property which is an important aspect of cache hierarchy design in modern microprocessors. In particular, we study cache bypassing in the context of inclusive last level caches. Cache bypassing (i.e. selective insertion) has been shown to outperform cache replacement policies with strict insertion policies [21][34]. Modern cache hierarchy designs adopt inclusive cache hierarchy which cannot incorporate cache bypassing due to fundamental conflict in the insertion policy. Therefore, we propose to solve this problem using a microarchitecture technique called ‘bypass buffer’ [25]. We present more detailed contributions and discussion in subsequent chapters.
Significant portion of the content presented in this dissertation has been published as mentioned in the following. Chapter 2 presents our detailed study on new model for locality and derives many insights using the model [26][27]. Chapter 3 uses the locality study to drive cache partitioning and presents the detailed implementation and experimental analysis of the proposed approach. Chapter 4 presents our adaptive cache bypassing research for inclusive last-level caches [25]. Chapter 5 summarizes our research contributions and concludes the dissertation with brief discussion on future work.
Chapter 2

Locality Principle Revisited: A Probability-Based Quantitative Approach

2.1. Introduction

The locality principle has been widely recognized and leveraged in many disciplines [13]. In computer systems, locality of references is the fundamental principle driving the memory hierarchy design. Specifically, the memory hierarchy relies on two types of locality, temporal and spatial locality. Temporal locality indicates that after an address is referenced, the same datum is likely to be re-referenced in the near future. Spatial locality, on the other hand, indicates that some neighbors of the referenced address are likely to be accessed in the near future. Given the significance of the locality principle, previous works have attempted to quantify the locality to better understand the reference patterns and to guide the compiler and architecture design to enhance/exploit program locality. Between the two types of locality, the histogram of reuse distances [76] or LRU (least recently used) stack distances [48] computed from an address trace is used as a measure of temporal locality. For spatial locality, however, there is a lack of consensus for such a quantitative measure and several ad-hoc metrics [4][6][24][68] are proposed based on intuitive notions. In this research, we revisit the concept of locality.
First, we propose to use conditional probabilities as a formal and unified way to quantify both temporal and spatial locality. For any address trace, we define spatial locality as the following conditional probability: for any address, given the condition that it is referenced, how likely an address within its neighborhood will be accessed in the near future. From this definition, we can see that the conditional probability depends on the definition of two parameters, the size of neighborhood and the scope of near future. Therefore, spatial locality is a function of these two parameters and can be visualized with a 3D mesh. Temporal locality is reduced to a special case of spatial locality when the neighborhood size is zero byte.

Second, we examine the relationship between our probability-based temporal locality and the reuse distance histogram, which is used in previous works for quantifying temporal locality. We derive that our proposed temporal locality (using one particular definition of near future) is the cumulative distribution function of the likelihood of reuse in the near future while the reuse histogram is the probability distribution function of the same likelihood, thereby offering theoretical justification for reuse distance histograms. The difference function of our proposed spatial locality over the parameter of the near future window size provides the reuse distance histograms for different neighborhood sizes. In comparison, the difference function of our proposed spatial locality over the parameter of the neighborhood size generates stride histograms for different near future window sizes.

Third, we use the proposed locality to analyze the effect of locality optimizations, sub-blocking/tiling in particular. The quantitative locality presented in 3D meshes visualizes the locality changes for different near future windows and neighborhood sizes, which helps us to evaluate the effectiveness of code optimizations for a wide range of cache configurations.

Fourth, we use the proposed locality to analyze and optimize cache hierarchies. We show that interesting insights can be revealed from the locality measure of access streams at different levels of memory hierarchy, which can then be used to drive the cache architecture optimization. We also showcase that our locality computation can be used to easily evaluate the effect of data prefetchers. Furthermore, we show how the locality correlates to the cache
replacement policy. Given the importance of last-level caches (LLCs) in multi-core architectures, we also present a detailed locality analysis to make a case for LLCs with large block sizes and relatively simple replacement policies.

Our proposed locality measure clearly presents the characteristics of a reference stream and helps to reveal interesting insights about the reference patterns. Besides locality analysis, another commonly used way to understand the memory hierarchy performance of an application is through cache simulation. Compared to cache simulation, our proposed measure provides a more complete picture since it provides locality information at various neighborhood sizes and different near future window scopes while one cache simulation only provides one such data point.

The remainder of this chapter is organized as follows. Section 2.2 defines our probability-based locality, derives the difference functions, and highlights the relationship to reuse distance histograms. Section 2.3 focuses on quantitative locality analysis of compiler optimizations for locality enhancement. Section 2.4 discusses locality analysis of cache hierarchies and presents how the visualized locality reveals interesting insights about locality at different levels of cache hierarchy and about data prefetching mechanisms. We also highlight how the locality information helps to drive memory hierarchy optimizations. Section 2.5 presents a GPU-based parallel algorithm for our locality computation. Section 2.6 discusses the related work. Finally, Section 2.7 summarizes the key contributions of this chapter.

**2.2. Quantifying Locality of Reference Using Conditional Probabilities**

**2.2.1. Temporal Locality**

Temporal locality captures the characteristics of the same references being reused over time. We propose to quantify it using the following conditional probability: given the condition that an arbitrary address, \( A \), is referenced, the likelihood of the same address to be re-accessed in the near future. Such conditional probability can be expressed as \( P(X_n = A, \exists n in near future \mid X_0 = A) \), where \( X_0 \) is the current reference and \( X_n \) is a reference in the near future.
future. Since an address stream usually contains more than one unique address, the temporal locality of an address stream can be expressed as a combined probability, as shown in Equation 1, where \( M \) is the number of unique addresses in the stream.

\[
TL = P(X_n = A, \exists n \text{ in near future } | X_0 = A)
\]

\[
= \sum_{i=1}^{M} P(X_n = A_i, \exists n \text{ in near future} | X_0 = A_i) \times P(X_0 = A_i)
\]  

Eq. 1

Equation 1 can be further derived using Bayes' theorem, shown as Equation 2.

\[
TL = \sum_{i=1}^{M} P(X_n = A_i, \exists n \text{ in near future } \cap X_0 = A_i)
\]  

Eq. 2

To calculate the joint probabilities, \( P(X_n = A_i, \exists n \text{ in near future } \cap X_0 = A_i) \), we need to formulate a way to define the term ‘near future’. We consider three such definitions and represent temporal locality as \( TL(N) = P(X_n = A, \exists n < N \mid X_0 = A) \), where \( N \) is the scope of the near future or the future window size.

Def. 1 of near future: the next \( N \) addresses in the address stream.

Def. 2 of near future: the next \( N \) unique addresses in the address stream.

Def. 3 of near future: the next \( N \) unique block addresses in the address stream given a specific cache block size. In other words, we ignore the block offset when we count the next \( N \) unique addresses.

Based on the definition of the near future, the joint probability \( P(X_0 = A \cap X_n = A, \exists n < N) \) can be computed from an address trace with a moving near future window. In a trace of the length \( S \), for each address (i.e., the event \( X_0 = A \)), if the same address is also present in the near future window (i.e., the event \( X_n = A, \exists n < N \)), we can increment the counter for the joint event \( (X_0 = A \cap X_n = A, \exists n < N) \). After scanning the trace, the ratio of this counter over \( (S - 1) \) is the probability of \( P(X_0 = A \cap X_n = A, \exists n < N) \). The reason for \( (S - 1) \) is to account for the fact that there are no more references (or any potential reuse) after the last one in the
During the scan through the trace, if def. 1 of near future is used, the moving window has a fixed size while def. 2 and def. 3 require a window of variable sizes. We illustrate the process of calculating temporal locality with an example shown in Figure 2.1.

Considering the trace shown in Figure 2.1, for each address (except the last one), we will examine whether there is a reuse in its near future window. If the near future window size is 2 and we use Def. 1 of near future, we will use a window of size 2 to scan through the trace as shown in Figure 2.1a. Based on the joint probabilities, the temporal locality, $TL(2)$, is $\frac{1}{4}$ as shown in Figure 2.1b. If we follow Def. 2 of near future, the temporal locality, $TL(2)$, is $\frac{2}{4}$ since each near future window will contain 2 unique addresses. If we use Def. 3 of near future and assume that $X$, $Y$, and $Z$ are in the same block, the temporal locality, $TL(2)$, is 1.

The temporal locality with various near future window sizes is shown in Figure 2.1c for these three definitions of near future.
2.2.2. Spatial Locality

Spatial locality captures the characteristics of the addresses within a neighborhood being referenced over time. We propose to quantify it as the following probability: given the condition that an arbitrary address, \( A \), is referenced, the likelihood of an address in its neighborhood to be accessed in the near future. Such conditional probability can be expressed as \( P(X_n \text{ in } A \text{'s neighborhood}, \exists n < N \mid X_0 = A) \), where \( X_0 \) is the current reference and \( X_n \) is a reference in the near future. Similar to temporal locality, the spatial locality of an address stream can be computed as the combined probability of all the unique addresses in the trace and it can be expressed as Equation 3, where \( M \) is the number of unique addresses in the stream.

\[
SL = P(X_n \text{ is in the neighborhood of } A, \exists n < N | X_0 = A) \\
= \sum_{i=1}^{M} P(X_n \text{ is in the neighborhood of } A_i, \exists n < N | X_0 = A_i) \times P(X_0 = A_i) \\
= \sum_{i=1}^{M} P(X_n \text{ is in the neighborhood of } A_i, \exists n < N \cap X_0 = A_i) \quad \text{Eq. 3}
\]

To compute the joint probabilities, \( P(X_n \text{ is in the neighborhood of } A_i, \exists n < N \cap X_0 = A_i) \), we need to define the meaning of the term ‘neighborhood’ in addition to near future. We consider two definitions for neighborhood and represent spatial locality as \( SL(N, K) \), where \( N \) is the near future window size and \( K \) is a parameter defining the neighborhood size.

Def. 1 of neighborhood: an address \( X \) is within the neighborhood of \( Y \) if \(|X - Y| < K\), where \( 2^*(K - 1) \) is the size of the neighborhood. The reason for \( 2^*(K - 1) \) rather than \( (K - 1) \) is to account for the absolute function of \((X - Y)\).

Def. 2 of neighborhood: two addresses \( X \) and \( Y \) are in the same cache block given the cache block size as \( K \). In other words, \( X \) is within the neighborhood of \( Y \) if \( X - (X \mod K) = Y - (Y \mod K) \).

Based on the definition of neighborhood, the spatial locality of a reference trace can be computed by moving a near future window through the trace, similar to temporal locality calculation. The difference is that for temporal locality, we require reuses of the same
addresses while for spatial locality, the addresses in the window are inspected to see whether there exists a reference $X_n, n < N$, falling in the neighborhood of $X_0$. If we change the definition of neighborhood size to zero (i.e., $K = 1$), spatial locality is reduced to temporal locality.

From Equation 3, we can see that spatial locality is a function of two parameters, the neighborhood size, $K$, and the near future window size, $N$. As such, we can visualize the spatial locality of an address stream with a 3D mesh or a 2D heat-map for different neighborhood and near future window sizes. Figure 2.2 shows the spatial locality, using the def. 2 of near future and def. 1 of neighborhood, of the benchmarks SPEC2006 mcf and SPEC2000 bzip2 (see the experimental methodology in Section 2.4.1). Temporal locality is the curve where the neighborhood size is zero.

Locality visualized in 3D meshes or heat-maps clearly shows the characteristics of reference streams. First, a contour in the 3D mesh/heat-map at a certain locality score (e.g., 0.9) can be used to figure out the sizes of an application’s working data sets. Comparing Figures 2.2b to 2.2d, it is evident that bzip2 has a much smaller working data set than mcf. We can also see that the size of the working set varies for different neighborhood sizes. For example, SL(8192, 32) of mcf is close to 0.9, indicating the working set size is slightly bigger than $2*32B*8192 = 512kB$ when the neighborhood size is 64 (= $2*32$) bytes. On the other hand, SL(65536, 16) of mcf is little less than 0.9, showing that the working set is greater than $(2*16B*65536) = 2MB$ when the neighborhood size is 32(=2*16) bytes.

Second, the quantified locality shows how it can be exploited. From Figure 2.2a, we can see that for mcf the reuse of the same data at the byte/word granularity (i.e., temporal locality) is limited even with a large near-future window. In contrast, there exists significant spatial locality that can be exploited with a moderate near future window, which also explains why the working set sizes vary significantly for different neighborhood sizes. Bzip2, on the other hand, shows much better temporal and spatial locality, which can be captured effectively with a relatively small neighborhood size. Note that the function $SL(N, K)$ is monotonous along either the $N$ or $K$ direction since our conditional probability definitions are
accumulative. In other words, a reuse within a small neighborhood (or close future) is also a reuse for a larger neighborhood (or not-so-close future).

Third, our proposed locality can be used to reveal interesting insights which are typically not present through cache simulations because one cache simulation only provides one data point in this 3D mesh. As one example, the benchmark, *hmmer*, shows steps in the locality scores (Figure 2.5a) when the near future size is at 32 and 8192, which implies that there is a good amount of data locality that we cannot leverage until we can explore the reuse distance of 8192. Such observations from the 3D mesh enable us to understand why and whether a particular cache configuration performs much better than the other ones.

Between heat maps and 3D meshes, locality presented in heat-maps better reveals the working set information. On the other hand, 3D-meshes are more useful in showing how the

Figure 2.2. The spatial locality plots of the benchmarks, *mcf* (SPEC2006) and *bzip2* (SPEC2000). Temporal locality is a special case of spatial locality with neighborhood size as 0.
locality changes along either the neighborhood size or the near future window size, which indicates how the locality can be exploited.

Different definitions of ‘near future’ lead to different near future window sizes as illustrated in Figure 2.1. Such differences, however, turn out to have little impact when we compute locality for most SPEC CPU benchmarks. Between the two definitions of neighborhood, although def. 2 models the cache behavior more accurately, it fails to capture spatial locality across the block boundary. For example, for a perfect stride pattern, 1, 2, 3, 4, 5, ..., the spatial locality using neighborhood def. 1 will be 1 for any neighborhood size and any future window size greater than 0. If we use def. 2 of neighborhood, the spatial locality will be dependent upon the neighborhood (or cache block) size. If the neighborhood/block size is 2 bytes, the spatial locality is 0.5 for any future window size. If the neighborhood/block size is B bytes, the spatial locality becomes \((B - 1)/B\) for any future window size greater than or equal to B. In Figure 2.3, we show the spatial locality for the benchmark *sphinx* with neighborhood def. 1 (Figure 2.3a) and def. 2 (Figure 2.3b). From the figure, we can observe the similar behavior, i.e., increasing locality values for larger neighborhood sizes in Figure 2.3b. In contrast, Figure 2.3a shows a stepwise increase in locality values. Based on these observations, we choose to use def. 2 of near future and def. 1 of neighborhood for locality analysis. Whenever the objective is to estimate the cache performance, def. 3 of near future and def. 2 of neighborhood can be used instead.

![Figure 2.3. Spatial locality of the benchmark sphinx for neighborhood definitions. (a) definition 1 (b) definition 2.](a) (b)
2.2.3. The Relationship between Spatial and Temporal Locality

Based on our probability-based definitions of spatial and temporal locality, temporal locality can be viewed as a special case of spatial locality. Beyond this observation, there also exist some intriguing subtleties between the two types of locality. In particular, we can change the neighborhood definition in spatial locality so that the exact same addresses are excluded from the neighborhood. In other words, the neighborhood definition becomes $0 < |X - Y| < K$. In this case, an interesting question is: for any address trace, is the sum of its spatial locality and its temporal locality always less than or equal to 1?

An apparent answer is yes since the reuses from the same addresses are excluded from the neighborhood definition. This is also our initial understanding until our locality computation results show otherwise. After a detailed analysis, it is found that the events for temporal and spatial locality are not always disjoint and therefore the sum can be larger than 1. To elaborate, we use def. 1 of near future and def. 1 of neighborhood with the modification that same addresses being excluded. Other definitions of the two terms can be applied similarly. Following Equations 1 and 3, temporal and spatial locality can be expressed as the following two conditional probabilities: $P(X_n = A, \exists n < N \mid X_0 = A)$ and $P(0 < |X_n - A| < K, \exists n < N \mid X_0 = A)$, respectively. Both probabilities can be computed using the sum of the joint probabilities $\sum_{i=1}^{M} P(X_n = A_i, \exists n < N \cap X_0 = A_i)$ and $\sum_{i=1}^{M} P(0 < |X_n - A_i| < K$,

![Figure 2.4. The Venn diagram showing the relationship among events $X_0 = A_I$, $X_I = A_I + \Delta$, and $X_2 = A_I$.](image)
\( \exists n < N \cap X_0 = A_i \). Since each \( A_i \) is unique, the events \( X_0 = A_i \) are disjoint for different \( A_i \). So, we can focus on one such address, e.g., \( A_i \). We can use the Venn diagram to capture the relationship among the events: \( X_0 = A_i, X_1 = A_i + \Delta, \) and \( X_2 = A_i, \) as shown in Figure 2.4. From Figure 2.4, we can see that \((X_0 = A_i \cap X_2 = A_i) \) and \((X_0 = A_i \cap X_1 = A_i + \Delta) \) are not disjoint. An intuitive explanation is that for an address sequence such as \( A_i, A_i+1, A_i, \) the third reference \((A_i)\) contributes to both temporal locality of the first reference and spatial locality of the second reference. Therefore, the events for temporal and spatial locality are not disjoint although the references to same addresses are excluded from the neighborhood definition.

### 2.2.4. Difference Functions of Locality Measures and the Relationship to Reuse Distance Histograms

Next, we derive the relationship of our probability-based temporal locality to reuse distance histograms, which are commonly used in previous works to quantify temporal locality. In sequential execution, reuse distance is the number of distinct data elements accessed between two consecutive references to the same element. The criterion to determine the ‘same’ element is based on the cache-block granularity, i.e., the block offsets are omitted when determining reuses. The reuse distance provides a capacity requirement for fully associative LRU caches to fully exploit the reuses. The histogram of reuse distances is used as the quantitative locality signature of a trace since it captures all the reuses at different distances.

Based on our probability-based measure, temporal locality \( TL(N) \) represents the probability \( P(X_n = A, \exists n < N \mid X_0 = A) \). As this probability is a discrete function of \( N \), we can take a difference function of temporal locality \( \Delta TL(N) = TL(N) - TL(N-1) = P(X_{N-1} = A \mid X_0 = A) \). This function represents the frequency of reuses at the exact reuse distance of \( (N - 1) \). If we use def. 3 of the term near future, the difference function becomes: \( \Delta TL(N) = P(X_{N-1} = block\_addr(A) \mid X_0 = A) \) where \( block\_addr(A) = A - (A \mod block\_size) \) and it is essentially the same as the reuse distance histogram. Therefore, we conclude that the reuse histogram is
the probability distribution function of the event \( (X_{N-1} = A \mid X_0 = A) \) and our proposed temporal locality is the accumulative distribution function of the same event.

As spatial locality, \( SL(N, K) \), is a function of two parameters, the near future window size \( N \) and the neighborhood size \( K \), we can compute the difference function of \( SL(N, K) \) over either parameter and both difference functions reveal interesting insights of the access patterns. Similar to temporal locality, the difference function of spatial locality over \( N \), \( SL(N, K) - SL(N - 1, K) = P( (X_{N-1} - X_{N-1} \mod K) = (A - A \mod K) \mid X_0 = A) \), shows the reuse distance histograms with different block sizes, \( K \), if we follow def. 2 of neighborhood.

**Figure 2.5.** Locality and the difference functions for the benchmark *hmmer*. (a) Spatial Locality (b) Difference function over the near future window size \( N \), which is the same as reuse distance histograms (c) Difference function over neighborhood size \( K \), which shows a histogram of stride access patterns.
The difference function over the neighborhood size parameter, \( K \), can be derived as follows: \( SL(N, K) - SL(N, K - 1) = P(|X_n - A| = K - 1, \exists n < N \mid X_0 = A) \). This probability function represents that in an address trace how often a stride pattern with a stride of \((K - 1)\) happens in a near future window of size \( N \). For different \( N \) and \( K \), this function essentially provides a histogram of different strides for any given near future window size. In the context of processor caches, this difference function helps to reason about the relationship among the cache sizes, which determine the maximum reuse distance or near future window size, the block sizes, which determine the neighborhood sizes, and the performance potential of stride prefectors.

In Figure 2.5, we present the spatial locality information (Figure 2.5a) of the benchmark, \textit{hmmer}, and its two difference functions (Figures 2.5b and 2.5c). From the Figure 2.5b, we can see how the reuse distance histograms vary for different block sizes. For small block sizes, many reuses can only be captured with long reuse distances. For large block sizes, the reuse distance becomes much smaller. This clearly indicates that one reuse histogram is not sufficient to understand the reuse patterns. Based on Figure 2.5c, we can see that for small near future windows, there are limited stride accesses. When the near future window size increases (larger than 32), the stride access pattern (stride of 8 bytes) is discovered. It implies that although the application has strong spatial locality in the forms of stride access patterns, in order to exploit such spatial locality, the cache size and set associativity need to be large enough to prevent a cache block from being replaced before it is re-accessed. In addition, the single dominant stride suggests that a stride prefetcher can be an effective way to improve the performance for this benchmark.

### 2.2.5. Sub-trace Locality

The locality defined in Equations 1 and 3 includes reuses of all the addresses in a reference stream. Therefore, it can be viewed as ‘whole-trace’ locality. We can add constraints to focus on certain reuses of interest, e.g., the locality of one memory access instruction or accesses to one data structure, within the address trace. We refer such locality to as sub-trace locality as not all the reuses in the trace will be considered. Note that sub-trace
locality is usually not the same as the locality computed from the filtered trace, e.g., the addresses generated by one instruction, because the filtered trace will miss the constructive cross-references from other instructions. Instead, sub-trace locality is computed from the ‘whole’ trace. Mathematically, sub-trace spatial locality (STSL) can be defined and derived as the sum of the joint probabilities as shown in Equation 4. Sub-trace temporal locality (STTL) can be derived similarly.

$$STSL = P(X_n is in the neighborhood of A, \exists n < N, X_n \in SetX \mid X_0 = A, X_0 \in SetY)$$
$$= \sum_{i=1}^{M} P(X_n is in the neighborhood of A_i, \exists n < N \cap X_0 = A_i \cap X_0 \in SetY \cap X_n \in SetX)$$

Eq. 4

From Equation 4, it can be seen that compared to whole trace locality, sub-trace locality is modeled with the additional joint events, $X_0 \in SetY \cap X_n \in SetX$, in the joint probability computation. The definition of $SetX$ and $SetY$ determines the event of interest. For example, to compute the locality of one particular memory access instruction, we can define $SetX$ as {the addresses generated by the instruction of interest} and define $SetY$ as the whole sample space $\Omega$ or {addresses generated by all instructions}, meaning that for all the addresses (i.e., $X_0 = A_i \cap X_0 \in SetY$), we need to check whether there is a spatial reuse in the near future by the instruction of interest ($X_n$ is in the neighborhood of $A_i, \exists n < N \cap X_n \in SetX$). If we change $SetY$ to $SetX$, the equation calculates the locality of the address trace generated by the instruction of interest (i.e., the filtered trace).

We can make a slight change to Equation 4 so that it can be computed more efficiently. Instead of looking for reuses in a near future window, we can use a near past window, as shown in Equation 5. This way, we can compute the locality of an instruction of interest as follows, for each address generated by the instruction (i.e., the event: $X_0 = A_i \cap X_0 \in SetX$), we search in a near past window for spatial reuse (i.e., the event: $X_n$ is in the neighborhood of $A_i, \exists n > -N, n < 0 \cap X_n \in SetY$.)

$$STSL = \sum_{i=1}^{M} P(X_n is in the neighborhood of A_i, \exists n > -N, n < 0 \cap X_0 = A_i \cap X_0 \in SetX \cap X_n \in SetY)$$

Eq. 5
Another use of sub-trace locality is to analyze the effectiveness of data prefetching. For example, consider the following address trace containing both demand requests (underlined addresses) and prefetch requests: A, A+32, A+64, A+96, A+128, A+64, A+128, A+160, A+192. In this case, we can define $SetX = \{\text{demand requests}\}$ and $SetY = \{\text{combined demand and prefetch requests}\}$. Then, sub-trace locality provides the information that for a demand request, how likely there are prefetch requests (or demand requests) for the same address in a near past window. The locality of either the demand request trace or the combined demand and prefetch request trace does not provide such information.

### 2.3. Locality Analysis for Code Optimizations

In this section, we use sub-blocking / tiling to show how our proposed measure can be used to analyze and visualize the locality improvement of code optimizations.

To analyze the locality improvement of the sub-blocking/tiling optimization, we choose to use the matrix multiplication kernel $C = A \times B$ and we use the pin tool [46] to instrument the binaries to generate the memory access traces for our locality computation. We used def. 1 of near future and def. 1 of neighborhood in our locality computation. Figure 2.6 shows the locality information collected for different versions of the kernel, including un-tiled, tiled with the tile size of 32x32, and the tiled with the size 64x16. The matrix size is 256x256.

In the un-tiled matrix multiplication kernel, the matrix A is accessed along each row in the sequence. Therefore, it exhibits spatial locality when the neighborhood size $\geq 4$ bytes, which is the size of each data element. Since the accesses to the matrices A and B are interleaved and B is accessed column-wise (i.e., no spatial locality at this neighborhood size), such spatial locality shows up when the near future window size $\geq 2$ and the locality score is 0.5. The kernel keeps using the same row from A until it is done with all the columns of matrix B. Consequently, after every 256 accesses along the row from A, the same elements are re-accessed. Again, due to the interleaved accesses of A and B as well as the accesses to the production matrix C, the reused distance actually is more than 512 (256 from matrix A + 256 from matrix B + 1 from matrix C) and hence the temporal locality appears at near future window size of 1024 in Figure 2.6a. Accesses to the matrix B do not repeat until 256x256x2
= 128k accesses, which is why temporal locality for the maximum near future value in Figure 2.6a is still around 50%. Since the matrix B is accessed column-wise, with the near future window size of 1024 and neighborhood size of 4, accesses to the matrix B start to enjoy spatial locality (i.e., B[i][j] and B[i][j+1] show in the same window) and the locality becomes very close to 1 as shown in Figure 2.6a.

With the loop tiling optimization, both matrices A and B are divided into tiles/sub-blocks. This reduces the reuse distance of the data elements as we can see from Figures 2.6b and 2.6c. For a tile size of 32x32, the inner loops perform matrix multiplication of two 32x32 matrices. As a result, the sub-blocks of A get reused when the near future window is larger than 64 and the sub-blocks of B get reused when the near future window is 2048 (= 32x32x2). The sub-blocks of matrix A are accessed row-wise and show 0.5 spatial locality when the neighborhood size is 4 bytes and near future window is smaller than 64. The sub-blocks of the matrix B show spatial locality when the near future window is larger than 64, which makes B[i][j] and B[i][j+1] show up in the same window. The 3D locality mesh shown in Figure 2.6b visualizes these changes in both temporal and spatial locality and confirms the observations. When the tile size is 64x16, the locality information, shown in Figure 2.6c, is the same as matrix multiplication of a 64x16 matrix with a 16x64 matrix.

Based on the locality information shown in Figure 2.6, we can easily see how much and where the locality improvement has been achieved with code optimizations. Furthermore, if
we use def. 3 of near future and def. 2 of neighborhood, we can also use our locality function to infer how much locality a specific cache can capture. For example, an 8kB cache with a 16-byte block size will be able to explore the future window up to 512 (8kB/16 bytes) and neighborhood range of 16 bytes. Therefore, SL(512, 16) will show the hit rate of this cache if it is fully associative. Mathematical models proposed in [8][31][58] can also be used to estimate miss rates for set-associative caches from the fully associative ones.

2.4. Locality Analysis for Memory Hierarchy Optimizations

Our proposed locality measure provides quantitative locality information at different neighborhood sizes and near future scopes. In this section, we show that it reveals interesting insights to drive memory hierarchy optimizations.

2.4.1. Experimental Methodology

In this section, we use an in-house execution driven simulator developed based on SimpleScalar [7] for our study, except stated otherwise in Section 2.4.4 and Section 2.4.6. In our simulator, we model a 4-way issue superscalar processor. The memory hierarchy contains a 32kB 4-way set-associative L1 data cache with the block size of 64 bytes (3 cycle hit latency), a 32kB 2-way set associative L1 instruction cache, and a unified 8-way set associative 1MB L2 cache with the block size of 64 bytes (30 cycle hit latency). The main memory latency is set as 200 cycles (224 cycles) to get a 64-byte (128-byte) L2 block. The memory bus runs at 800MHz while the processor core runs at 2GHz.

We include all the SPEC CPU2000 (SPEC2000) [82] and SPEC CPU2006 (SPEC2006) [81] benchmarks that we were able to compile and run for SimpleScalar ISA (PISA), 16 from SPEC 2000 and 8 from SPEC 2006. Benchmarks mcf and bzip2 from SPEC2006 are abbreviated as mcf-2006 and bzip2-2006. For each benchmark, we use a representative 100M Simpoint [29] for our trace collection and simulations. In our locality computation, we use the def. 2 of near future and def. 1 of neighborhood.
2.4.2. Locality at Different Memory Hierarchy Levels

In this experiment, we examine the locality of the L1 data cache access trace and the L2 cache access trace. We use the benchmarks art, mcf (SPEC2000) and bzip2 (SPEC 2000) as our case studies. Figure 2.7 shows the locality of L1 and L2 access traces for art, bzip2, and mcf.

By comparing Figures 2.7a to 2.7b, 2.7c to 2.7d and 2.7e to 2.7f, we can see that the L1 cache effectively exploits the spatial locality with neighborhood size less than 64 bytes (K =

![Graphs showing locality at different memory hierarchy levels for art, mcf, and bzip2 benchmarks.](image)

Figure 2.7. The spatial locality of the benchmarks, art, mcf (SPEC 2000), and bzip2 (SPEC2000) at different cache levels.
32), which is the L1 block size. Also, the capacity of the L1 cache enables it to exploit a near future window of 512 (=L1 size/block size). As a result, at the L2 level, there is limited locality present in this range. So, in order for L2 to become effective, it has to support much larger near future windows (by a large L2 cache size) and/or larger neighborhood sizes (by a large L2 block size). For the benchmark \textit{art}, we can see from Figure 2.7a that the locality does not increase much as we increase the near future window size up to 32768. If we keep the L2 cache block size the same as the L1 block size, the gains by increasing the cache size is very limited until it can explore the near future window size of 32768 (implying a L2 cache size of 32768 x 64B= 2MB) as shown in Figure 2.7a. While making the cache block size 128B will significantly improve the spatial locality exploited by L2 cache. For the benchmark \textit{mcf}, similar observations can be made from the locality curve. For the benchmark \textit{bzip2}, the spatial locality of the L2 accesses is lower than the other two benchmarks in Figure 2.7.

At lower levels of caches in a cache hierarchy, the accesses seen are primarily the misses from upper levels. This causes the lower levels to have no spatial locality until they have block sizes bigger than upper levels.

2.4.3. Memory Hierarchy Optimizations

1) Optimizing Last-Level Caches (LLCs):

In order to improve the performance of the last-level cache, the L2 cache in our simulator, we examine the locality information of the L2 access stream. The L2 access locality of \textit{art} and \textit{mcf} is shown in Figures 2.7b and 2.7d. In Figure 2.8 and 2.9a, we report the locality information of L2 access streams of the SPEC benchmarks, \textit{sphinx, ammp, milc, mesa, gcc, equake, vortex, swim} and \textit{wupwise}. Two important observations can be made from the locality results. First, a common theme of all the locality information is that there exists strong spatial locality when the neighborhood size is larger than 32 bytes, corresponding to a cache block size of 64 bytes given our neighborhood definition 1. The behavior is expected as the L1 cache block size is 64 bytes.
However, in many current commercial processors, the same cache block size (e.g., 64 bytes in Intel Core i7) is used for different level of caches to simplify the cache coherence.

Figure 2.8. The spatial locality of L2 access traces for various benchmarks.
management. From the locality information shown in Figure 2.8 and Figure 2.9a, we can see that many benchmarks require much larger neighborhood sizes to take advantage of the spatial locality. Among these benchmarks, wupwise, milc, mesa, equake, vortex and gcc would need the neighborhood size of 64 bytes (i.e., 128-byte cache block) while sphinx, mcf and art exhibit additional spatial locality for increasing neighborhood sizes. The benchmark, ammp, requires a neighborhood size of 2048 bytes due to its streaming access pattern with a stride of more than 1024 bytes.

Second, for all these benchmarks, the spatial locality can be exploited within a small near future window (or reuse distance) when a large block is used. This indicates that a small cache with relatively large block sizes can be more effective than a large cache with small block sizes. For example, for benchmark gcc, SL(4,64) = 0.97 (corresponding to a 4x128 = 512 byte cache with the 128-byte block size) is higher than SL(8192,32) = 0.008 (corresponding to a 8192 * 64 = 512KB cache with a 64-byte block size) and is very close to SL(16384, 32) = 0.99 (corresponding to a 16384*64 = 1MB cache with a 64-byte block size). Since most of the locality can be captured within a small near future window, it also means that the LRU replacement policy will work well for caches with large blocks, thereby eliminating the need for more advanced replacement policies (more discussion in Section 2.4.6). Furthermore, the limited near future window indicates that the required set associativity is small as well.

Different approaches can be used to achieve the effect of large-block caches. The first method is to directly increase the cache block size. The advantage is that it can reduce the required cache size as discussed above. The downside is that it may not work for all applications because different applications prefer different block sizes. The second method is to emulate the big block size by augmenting the cache with next-n-line / previous-n-line prefetching. The way to determine the value ‘n’ in next/previous-n-line prefetching is based on our locality measure. With the current block size of 64 bytes, if there is high spatial locality at block size of 128 bytes, e.g., gcc, next/previous line prefetch is used (i.e., n=1). If the spatial locality shows up at a much larger block size, e.g., 2kB for ammp, n is set as
$2kB/64B = 32$. In other words, our locality measure can be used a profiling tool to control the next/previous-n-line prefetching. The advantage of this approach is that it can emulate different cache block sizes without changing the cache configurations (i.e., cache size, set associativity, and block size). The third way is to use a stream buffer to detect stride patterns and prefetch data blocks into caches.

![Figure 2.9. The spatial locality for wupwise (a) L2 access trace (b) L2 miss stream for 1MB L2 cache with a 64B block size (c) L2 miss stream for 1MB L2 cache with a 128B block size (d) L2 miss stream with a stream buffer.](image)

Next, we evaluate and compare the effectiveness of these different approaches using our locality measure with a case study of the benchmark *wupwise*. Here we use the locality measure of the L2 demand miss stream as it shows the locality that the L2 cache fails to exploit. In Figures 9b, 9c, and 9d, we show the L2 miss locality measure for *wupwise* when the L2 cache uses a 64-byte block size, a 128-byte block size, and a stream-buffer. The next/previous line prefetch has very similar results to the cache with the 128-byte block size.
From Figure 2.9b, we can see that the L2 cache with a 64-byte block size fails to exploit the spatial locality at large neighborhood sizes. In contrast, Figure 2.9c shows that the 128-byte block cache with the same size exploits the spatial locality effectively. The stream prefetcher does not work as well for wupwise as the stride access patterns are not common in this benchmark (Figure 2.9d). Our timing simulation also confirms the similar trend in performance improvement for wupwise: an execution-time reduction of 5% using the stream buffer while the 128-byte block size and next line prefetching achieve the execution-time reduction of 11% to 14%, respectively.

Figure 2.10 shows performance improvements measured in IPC (instructions per cycle) speedups for large block sizes, next/previous-n-line prefetching, and stream buffer prefetcher. Among the benchmarks, equake, gap, hmmer, mcf(SPEC2000), mcf (SPEC2006), milc, parser, perl, and wupwise favor large block sizes (or previous/next-n-line prefetching). On average, 4.7% (6.6%) IPC improvement is achieved on average across all the benchmarks using large block sizes (previous/next-n-line prefetching). Benchmark swim experiences a slowdown when bigger block size or next/previous-n-line prefetching is used. The results can be explained with the locality of its L2 access trace (Figure 2.8i). From the figure, we can see that the difference in locality score, when the block size is changed from 64 bytes to 128 bytes, is very small for swim. Therefore, the increased latency to get a larger
data block results in a slowdown. Benchmarks ammp, art, gap and lbm have stride beyond 64 bytes, therefore 128 byte block size is not much helpful for these benchmarks. We determine the more suitable ‘n’ value to be used in next/previous nth-line prefetcher from the locality measure. For most benchmarks, the next/previous nth-line prefetching successfully emulates the bigger cache block size. On the other hand, the benchmarks ammp, bzip2(SPEC2000), bzip2(SPEC2006), gcc, gromacs, lbm, sphinx, swim and vortex benefit more from the streaming buffer and it shows 17% performance gains on average across all the benchmarks.

2) Optimizing Memory Controller:

Even with the spatial locality being leveraged by large blocks or stream buffers, when we examine the locality of the L2 demand miss stream, we observe that there is residual locality to be leveraged. As an example, Figure 2.11a shows the locality information for the demand memory access stream of the benchmark equake when the L2 cache has the cache line size of 128 bytes. From the locality measure, it can be seen that there exists strong spatial locality

Figure 2.11. The spatial locality of L2 miss stream for the benchmarks (a) equake; (b) lbm; (c) mcf (SPEC2000); and (d) milc.
when the neighborhood size is larger than 256 bytes ($K = 128$). Although DRAM system can exploit some of this locality in the DRAM row-buffer, where a row-buffer hit has significantly lower latency than a row-buffer miss, our locality measure reveals further insights on how to better exploit the locality. From Figure 2.11a, we can observe that with a short near future window of 1 reference, the locality score $SL(1, 256)$ is 0.29 for *equake*. When increasing the near future window size to 4 references, the locality score $SL(4, 256)$ for *equake* is increased to 0.54. Similarly, the benchmarks, *lbm, mcf* (SPEC2000) and *milc*, have significantly higher locality for near future window sizes larger than 1 (Figure 2.11b, 2.11c and 2.11d). Based on this observation, we propose to add a small buffer at the memory controller level. This buffer has large block sizes (e.g., 1kB to 4kB) but with just a few entries (e.g., 4).

Compared to the row buffer in the DRAM, which can be viewed as an off-chip single-entry buffer with a block size of 4kB, it has two benefits: (1) a multi-entry buffer is able to exploit the significantly higher locality present beyond the near future scope of 1 reference, and (2) a buffer located at the on-chip memory controller has much lower latency than the off-chip DRAM row buffer. For the benchmarks showing no such spatial locality, we can simply power-gate this small buffer.

![Figure 2.12. Performance improvements from the proposed buffers at the memory-controller level.](image)
In our experiment, we use a 4-entry buffer with each entry caching 1024 bytes of data. A 10-cycle hit latency is assumed for this buffer, which is accessed after an L2 cache read miss. This buffer uses a write-through write-not-allocate policy for write requests. For comparison, we also model a single-entry 4kB buffer with the same hit latency. In order to model memory system more accurately, we also take into account the row buffer locality present in DRAM system by modeling a 100 cycle latency of a row-buffer hit as compared to a 200 cycle latency when there is a row-buffer conflict.

Figure 2.12 shows performance results of our timing simulations. From the figure, we can see that many benchmarks benefit from caching the data in a buffer at the memory controller level. Between the two buffer designs (4-entry buffer with a block size of 1kB vs. single-entry buffer with a 4kB block size), the benchmarks *art, equake, gap, lbm, mcf* (SPEC2000), *milc* and *sphinx* show higher performance when the 4-entry buffer is used. For *ammp*, the single-entry buffer has higher performance due to its near perfect stride access pattern with the stride larger than 1kB. On average, a 4kB single-entry buffer improves performance by 7.8% while a 4-entry buffer with the block size of 1kB improves the performance by 10.3% across these benchmarks.

### 2.4.4. Locality Improvement from Data Prefetching

As discussed in Section 2.5, we can use our proposed sub-trace locality measure to examine the locality improvement of data prefetching mechanisms. With the locality scores present for various neighborhood sizes and near future window scopes, we can tell how well a prefetcher under study works in a wide range of cache configurations.

In this experiment, we use the simulation framework for JILP Data Prefetching Contest [79] and select one of the top performing prefetchers [14] to illustrate how to interpret its effectiveness from our locality measure.

We examine two benchmarks, *mcf* and *soplex*, as either enjoys significant performance gains from the prefetcher (CPI reduced from 3.17 to 0.71 for *mcf* and from 0.61 to 0.43 for *soplex* when the L1 cache is 32kB and L2 cache is 2MB and the block size is 64B) and exhibits interesting locality behavior. As the data prefetcher under study is an L1 cache
prefetcher, we examine the locality improvement at both the L1 and L2 caches. For either benchmark, the locality of the L1 (demand) access stream without the prefetcher and the sub-trace locality of the L1 demand accesses within the combined demand and prefetch stream show very similar locality pattern and the improvement is limited (less than 0.05 in the locality score or 5% in L1 cache hit rate). This shows that the performance improvements are not from the locality improvement at the L1 cache level. In contrast, the locality at the L2 cache level is highly improved, revealing the reason for the significant performance gains. In Figure 2.13, we compare the locality of the L2 demand access stream (Figures 2.13a, 2.13c)
without the prefetcher to the sub-trace locality of the L2 access stream with the prefetcher being enabled (Figures 2.13b and 2.13d) for mcf and soplex.

Figure 2.14. The locality improvement at L2 cache from data prefetcher for milc, hmmer, lbm and calculix shown by comparing the locality of demand access trace and sub-trace locality of demand accesses with prefetcher
From Figure 2.13, we can see that for soplex the locality is improved for small near future windows (from near future window = 1 onwards in Figure 2.13d) which implies that the prefetches are issued very close to the demand accesses. For mcf, the locality improvement shows up for after near future window of 32. This explains why the preftrch is more effective for mcf compared to soplex. Also, the locality improvements for a zero byte neighborhood size show that the preftrcher is able to predict the byte addresses accurately sometimes. Another interesting observation from Figures 2.13(b) and 2.13(d) is that even with a quite effective data preftrcher, large neighborhood sizes still show higher locality scores. It indicates that although the data preftrcher effectively exploits certain types of access patterns such as strides, there exists spatial locality, which is not captured by the preftrcher but can be leveraged with large cache block sizes.

For some additional benchmark applications, milc, hmmer, lbm and calculix, Figure 2.14 compares locality of demand accesses with sub-trace locality of demand accesses with preftrcher. We can observe that for milc and hmmer, there is a slight improvement in locality for the region where the L2 cache configuration lies. However, lbm and calculix show significant improvement in locality. The key difference in locality improvements lbm and calculix is that for lbm, the locality improvement is mainly for the long reuse distances (i.e. corresponding to the sizes of L2 capacity) and locality of short near future window sizes is dropped. This is due to prefetches being inserted into near future window that show reuse at a later time for larger window sizes. Similar to mcf, it is indicative of better timing and effectiveness of prefetching for lbm. On the other hand, calculix shows significant improvement in spatial locality at short reuse distance (near future window of 2) as well as temporal locality improvements for long reuse distances (temporal locality for short reuse distance is lowered but long reuse distance is increased). The improvement in spatial locality at short distance indicates that some prefetches are in the neighborhood of demand accesses but they may see a demand access corresponding to these prefetches after a while (temporal locality improves for near future window sizes beyond 1024). Appendix B presents sub-trace locality for some additional benchmarks as well as compares locality of L2 access trace vs. L2 miss trace for selected benchmarks (that have significant number of L2 misses).
2.4.5. Understanding the replacement policy

It has been well understood that some applications work well with the least recently used (LRU) replacement policy while others do not. We examine this question using our probability-based locality measure. We first focus on L1 cache access streams. Figure 2.7e (or Figure 2.2c) shows that with block size larger than 4 bytes, bzip2 features a high amount of locality within small near future windows. For larger windows, the locality improvement is pretty much saturated. This implies that the L1 access stream of bzip2 is LRU friendly as most of the data or their neighbors will be used in a ‘short/small’ near future window. In comparison, if we look at the locality of the L1 accesses for mcf (SPEC2006), shown in Figure 2.2a, the locality is very gradual when the block size is less than 4 bytes and there is no “knee” behavior along the X axis (the near future window size) as has been observed in bzip2. This shows that in mcf, many data reuses require long reuse distance, which is not LRU friendly as they are likely to become least recently used before they are re-accessed. Interestingly, if we look at the other dimension of the 3D mesh for mcf, we can see that it becomes more LRU friendly for as neighborhood size goes up (i.e., when the cache block sizes $\geq$ 4 bytes).

Next, we examine the L2 access streams. As discussed in Section 2.4.2, the L1 cache effectively exploits the locality in small near future windows and the L2 cache has to explore a larger near future window to capture the remaining locality. From Figure 2.7f, we can see that for bzip2, with the neighborhood size being 64 bytes ($K = 32$), once the window becomes reasonably large (e.g., 16384, corresponding to a 1MB = 16384*64 B cache), there exists significant locality. Therefore, it is still LRU friendly. In contrast, the locality is not present for small near future windows for benchmarks art or mcf (SPEC 2000) (Figure 2.7b and 2.7d). This suggests that for art or mcf, there are many reuses at the L2 level requiring very long reuse distances. For an L2 cache with limited capacity (less than 2MB), the LRU replacement policy does not work well since the cache is not able to sustain such large reuse distances and exhibits thrashing behavior. For such access patterns, thrash resistant replacement policies such as DIP [51] or DRRIP [34] can be a better choice. The locality plot
also reveals that it gets more reuse (or becomes more LRU friendly) if the cache block size is larger than 64 bytes.

2.4.6. Spatial Locality vs. Temporal Locality in Last Level Caches

Given the importance of last-level caches (LLCs) in multi-core architectures, we have observed a resurgence of cache management research work toward LLCs, such as different intelligent cache replacement policies. In this section, we present a detailed locality analysis for LLCs, revisit these newly developed cache replacement policies, and make a case for LLCs with large block sizes but relatively simple replacement policies.

We adopt a different methodology in this section to simulate different cache replacement policies under a common framework provided for Cache Replacement Championship (CRC) from JWAC-1 [80]. The processor model has an 8-stage, 4-wide out-of-order pipeline with a 128-instruction scheduling window. The memory hierarchy contains 3 cache levels: a per-core private 32KB L1 data cache (8-way set assoc. 64-byte cache line, 1-cycle hit latency) and a per-core private 32KB L1 instruction cache (32KB, 4-way set assoc. 64-byte cache line, 1-cycle hit latency), a per-core private 256KB L2 cache (8-way set assoc. 64-byte cache line, 10-cycle hit latency), and a shared 1MB/4MB LLC (16-way set assoc. 64-byte cache line, 30-cycle hit latency, 200-cycle miss latency). Note that in this framework as well as in practice, the cache block sizes are kept the same across various cache levels because it greatly simplifies the coherence protocol among them. This, however, causes the LLC to have a limited opportunity to exploit spatial locality, as discussed in Section 2.4.2. As a result, the temporal locality captured by LLCs is determined mostly by their capacity since set-associative caches already reduce conflict misses by a large amount. Therefore, LLCs needs a significantly higher capacity to capture long reuse distances for them to be effective. The locality plots shown in Figure 2.15 confirm this observation by comparing locality of xalancbmk and milc. Appendix A shows locality plots for several other SPEC2006 benchmarks that we studied.

From Figure 2.15a, we can see that for the benchmark, xalancbmk, with the neighborhood size being 64 bytes (K = 32), after the window becomes reasonably large (e.g.,
16384, corresponding to a 1MB =16384*64 B cache), there exists significant locality. This implies that a high fraction of the accesses observe a reuse within the window that can be captured by the LLC and therefore, it is not thrashing in nature for near future window sizes bigger than 16384. In contrast, for the benchmark, milc (Figure 2.15b), the locality for near future windows of 16384 and beyond remain very low when the neighborhood size is kept as 64 bytes (K=32). This suggests that for LLC, there are many reuses requiring very long reuse distances, which translate into a requirement of capacity much higher than 1MB (=16384*64B). As a result, the LRU replacement policy does not work well for an LLC with limited capacity. On the other hand, it is very interesting to see from the locality plot that milc gets more reuses (or becomes more LRU friendly) if the cache block size is larger than 64 bytes. In other words, the same access pattern can be LRU-friendly or LRU-unfriendly/thrashing depending upon the size of neighborhood exploited in the cache design.

In the next experiment, we compare the performance gains from exploiting temporal locality using intelligent replacement policies with performance gains from exploiting spatial locality using bigger cache block sizes. Figure 2.16 presents the IPC speedup of the different benchmarks when they are simulated with different LLC configurations. We vary the cache block size as well as alter the cache replacement policy to see the impact of different replacement policies with different cache block sizes. On average, the more intelligent replacement policies including 3P-4P [49] and DRRIP [15] can improve the performance by
6% and 11% on average w.r.t. LRU when 64-byte cache blocks are used. In comparison, increasing the cache block size to 128-byte and 256-byte can provide IPC speedup of 10% and 37% on average, respectively using the LRU replacement policy. It is interesting to see that performance difference between intelligent replacement policies and LRU replacement policy is less than 2% when 128-byte/256-byte block sizes are used. The only exception among all the benchmarks is xalancbmk, for which the locality score SL(16384, 32) = 0.76 while SL(8192, 64) = 0.54 and SL(4196, 128) = 0.57, meaning that for this benchmark, increasing the block size while keeping the same overall cache capacity (1MB) results in less locality.
Overall, our observations from locality plots are confirmed as follows: (1) locality plots suggest that the benchmarks become more LRU friendly when cache block sizes are increased, and (2) as the locality curve is not LRU friendly at smaller block sizes, intelligent replacement policies are effective when 64-byte cache blocks are used, and (3) with bigger cache block sizes they become LRU friendly and intelligent replacement policies render to be less effective compared to LRU. We also repeat the same experiment on a 2MB LLC. The results are shown in Figure 2.16b, which shows a very similar trend to Figure 2.16a.

To get better understanding why increasing block sizes is highly effective in capturing locality for LLCs, it is important to highlight that the higher locality is not only a result from reduced compulsory misses, but also from reduced capacity misses [31]. This can be illustrated with an example access pattern. Consider the access pattern, A, A+64, A+128, A+192, A, A+64, A+128, A+192. For an LRU managed, fully associative cache with 64-byte blocks and capacity of 128 bytes, there are 4 compulsory misses and 4 capacity misses. The last 4 accesses are misses due to limited capacity. When the block size is 128-bytes, there are 2 compulsory misses and 2 capacity misses. In other words, both compulsory and capacity misses are reduced as a result of larger block sizes.
We also quantitatively analyze the reduction in miss-rate for each type of misses in the benchmarks. Figure 2.17 presents the contribution of each type of misses to the overall miss-rate as a stacked column plot. Most benchmarks encounter a small fraction of conflict misses due to high set-associativity of LLC (16). Among the benchmarks we studied, only bzip2 and libquantum have more compulsory misses as compared to capacity misses. All other benchmarks have high amount of capacity misses. When the cache block size is increased from 64-byte to 128-byte we see a reduction in compulsory misses as well as capacity misses for all the benchmarks except xalancbmk. For xalancbmk, although the compulsory misses are reduced, its capacity misses are increased due to cache pollution.

As the results shown in Figure 2.16 are based on a single workload utilizing the LLC, in the next experiment, we evaluate the performance gains of exploiting temporal locality using intelligent replacement policies with performance gains of exploiting spatial locality using bigger cache block sizes when LLCs are shared among multiple cores. Here, for our multicore evaluations, we create 16 multi-programmed workloads by randomly picking 4 benchmarks to combine in mixes. These 4-way multi-programmed workloads are executed on a 4-core configuration with a shared L3 cache of 4MB capacity. We use instructions per cycle (IPC) for comparing performance in single-core scenario and weighted IPC speedup, i.e. \( \sum_i (\frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{alone}}}) \), for comparing the performance in multicore scenario. This

Figure 2.18. Weighted speedups of LRU, 3P-4P and DRRIP with different cache block sizes (4MB LLC) over the baseline 4MB LLC with a 64-byte block size and the LRU replacement policy.
methodology of evaluation is adopted from the 1st JILP-CRC. The results are shown in Figure 2.18.

From Figure 2.18, we observe that for LLCs shared among multi-programmed workloads, 3P-4P and TA-DRRIP can provide good speedups when the LLC block size is 64-byte. When the cache block size is 128 or 256 bytes, in contrast, they have similar performance to the LRU replacement policy on average. Therefore, these results are consistent with those shown in Figures 2.15 and Figure 2.16 and further strengthen our argument for LLCs with large block sizes and relatively simple replacement policies.

2.5. A GPU-based Parallel Algorithm for Locality Computation

From the discussion in Sections 2.3 and 2.4, we show that our probability-based locality provides a useful way to analyze reference patterns and drive memory hierarchy designs. However it requires a significant amount of computation time, particularly for a large trace as we need to move a near future window throughout the trace. As derived in Section 2.2.4, our temporal locality essentially represents the same information as reuse distance histograms (cumulative distribution function vs. probability distribution function). Although we can leverage the previous works on fast computation of reuse distances [3], we aim to reduce the computation time by an order of magnitude to make it practical for compiler or runtime profile analysis. Based on the inherent data-level parallelism of our locality computation, we resort to parallel computation on graphics processing units (GPUs).

Our parallel algorithm for locality computation of an address trace $A[0 : S-1]$ is outlined as follows.

1. Each thread will be responsible to the locality information of $A[thread id]$. More specifically, it calculates the joint probability $P(X_n \text{is in the neighborhood of } A[thread id], \exists n < N \cap X_0 = A[thread id])$ for a specific neighborhood size $K$. It does so by comparing $A[thread id]$ with $A[thread id + 1], A[thread id + 2], A[thread id + 3]$, etc., until it exceeds the maximal near future window size or finds an address meeting the requirement of neighborhood. We keep $W$ counters in each thread for $W$ different near future window sizes. When an address in its neighborhood is found, the
distance to $A[thread id]$ is determined. If the distance is smaller than a near future window size, the corresponding counter is incremented.

2. After we obtain the locality information of each address, we need to combine them to generate the locality information of the trace. Since we have $W$ counters in each thread, we will perform $W$ reductions across all threads to accumulate them. After reduction, those accumulated counters are divided by $S$ to generate the data points of the locality function, i.e., $SL(1, K), SL(2, K), SL(4, K), \ldots, SL(2W, K)$.

3. Repeat steps 1 and 2 for a different neighborhood size $K$.

We implement this algorithm using CUDA [78]. In our implementation, both steps 1 and 2 are optimized by utilizing GPU’s on-chip shared memory. We run our GPU algorithm on a NVIDIA GTX480 GPU and compare with our CPU implementation on an Intel Core i7 920 processor. We compute locality with the near future window size ranging from 1 to 216 addresses (exponential scale) and we vary the neighborhood size from 0 to 512 bytes (exponential scale). For address traces with the length of 1 million to 16 million addresses, our GPU implementation takes less than 4 seconds and achieves 30X to 33X speedups over our CPU implementation.

2.6. Related Work

Given its importance, the principle of locality has been extensively studied. However, most previous works on quantifying locality, spatial locality in particular, are based on intuitive notions or heuristics and there is a lack for formal quantitative definition of locality.

For temporal locality, reuse distances or LRU stack distances [48] are used to determine how far in the future a temporal reuse is going to happen. The histogram of reuse distances is used as a signature to quantify the temporal locality of a trace [76]. StatCache [5] provides an efficient way to estimate reuse distance histograms through sampling. Section 2.2.4 derives the relationship between our probability-based measures and reuse histograms and provides a theoretical justification for reuse histograms. Based on this relationship, previous works on reuse distances such as component-based locality analysis [15][75] can also be adapted to
our proposed measure. In [19], mass-count disparity is used to show that most reuses are from accessing a small number of addresses.

For spatial locality, Bunt et al. [6] assume that the references can be fit to a Brad-Zipf distribution and the parameters of such a distribution are used to generate a spatial locality score. Weinberg et al. [68] propose to compute a histogram of different strides within a window of previously accessed addresses and then combine the histogram with strides into a single score for spatial locality. Berg et al. [4] and Gu et al. [24] propose to quantify the spatial locality effect using the changes in the miss rates or the reuse histograms when the block size is increased or doubled.

Using a 3D surface to visualize locality is proposed in [23] and it is recognized that the temporal locality is a special case of spatial locality. The locality measure in is based on an averaged probability that the next access is within a stride of the current location within a near future window, which is fundamentally different from our proposed conditional probability.

Compared to previous attempts on quantifying temporal or spatial locality, the advantages of our proposed approach include: (1) it provides a numeric measure with a clear mathematical meaning (i.e., conditional probability); (2) it offers a unified quantification for both temporal and spatial locality; and (3) it does not only present a single locality score for a specific cache configuration but also shows the trend of how locality vary when we change the near future window and neighborhood size so that we can better understand the nature of the program locality.

2.7. Summary

The locality principle describes the phenomenon that after a reference to a datum, the same datum or its neighbors are likely to be referenced in near future. Our proposed locality measure addresses the question ‘how likely’. The key contribution of this research includes (a) a formal definition of locality as a conditional probability and derivation of how it can be computed using joint probabilities, which can in turn be computed from address traces, (b) the derivation of difference functions of the locality function and the justification for reuse
histograms as a quantitative measure of temporal locality, (c) the formal definition of sub-
trace locality to focus on certain reuses of interest, and (d) various locality analysis using our
proposed measure, revealing the following insights (1) how locality can be exploited at
different levels of memory hierarchy and how the locality curves drive various memory
optimizations; (2) the information of the working set: instead of the common way of using
the knee of the miss rate curve over cache sizes, the working set is represented as a contour
in a 3D surface showing the same conditional probability can be exploited using different
neighborhood sizes and near future scopes; (3) an evaluation of memory optimizations such
as data prefetcher over a wide range of cache configurations; and (4) a detailed locality
analysis for LLCs to make a case for LLCs with large block sizes and relatively simple
replacement policies.
Chapter 3

Locality Driven Cache Hierarchy Optimizations for Energy Efficient and High Performance Caching

3.1. Introduction

Cache management policies have significant impact on cache performance as well as overall system performance. Caches exploit temporal and spatial locality and therefore locality characteristics are the fundamental choice to guide cache management. In the Chapter 2, we have proposed the probability based locality measure which can be used to drive different cache optimizations. Dynamic reconfiguration of a private cache and dynamic capacity partitioning of a shared cache have been studied before and have been shown to improve energy efficiency and/or performance. In previous works, temporal locality is characterized using working set analysis [16] or reuse distance analysis [48] while spatial locality is characterized/estimated by ad-hoc techniques [6][24][68]. In this chapter, we leverage our unified definition of locality and propose an online locality monitoring framework to drive cache management policies. The effectiveness of our proposed algorithms is shown on last level caches (LLCs) in both single-core and multi-core processors.
To motivate our algorithms first we revisit the locality plot for the level-3 (L3) cache reference stream of benchmark gcc, shown in Figure 3.1, as an example. In the figure, the block based definition of neighborhood and near future is used to compute spatial locality.

As discussed before, locality plot shown in Figure 3.1 provides the hit rates for various cache configurations. For example, for the commonly used cache block size of 64 Bytes (B), the locality score is close to zero when the near future window size is less than 8192. When the future window size is 16384, the locality $SL(16384, 64)$, corresponding to the hit rate of a 1MB (=64B*16384) L3 cache with a block size of 64B, is 16.5%. In comparison, the locality score $SL(8, 512)$, corresponding to a 4kB (=512B*8) L3 cache with a cache block size of 512B, is 76.8%. This shows that with a large cache block size, a small cache can outperform a much larger cache with a small block size. This highlights the fact that the cache capacity requirements (or the working set size) are highly dependent on the block size and this fundamental observation is the key driving factor of our cache management schemes and distinguishes our work from others. Moreover, this example showcases that working set analysis focused on temporal locality only provides only the partial picture of locality and hence a unified analysis of both spatial and temporal locality is a must in driving cache management.

![Figure 3.2: The locality plot of the benchmark gcc at L3 level.](image)

We further validate our observation with simulation results of selected memory-intensive benchmarks. Their performance, measured in instructions per cycle (IPC), is presented in
Table 3.1 for different cache configurations (see the methodology in Section 3.5). The second column in Table 3.1 presents the performance results, including IPC and the miss rates measured using the number of misses per kilo instructions (MPKI), for a baseline 16-way set-associative 1MB cache with a 64B block size. The third column presents low-capacity cache configurations, labeled as (capacity, block size), determined from the locality plots for each benchmark, which provide a similar or higher locality score than the baseline. The set associativity remains as 16. The fourth column presents the performance of these cache configurations. From Table 3.1, we can see that for these benchmarks, small-capacity caches with large block sizes can achieve similar or superior performance to large-capacity ones with a small block size of 64B.

Table 3.1: Performance comparison among different cache configurations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline Performance IPC (MPKI)</th>
<th>Low Capacity Cache Configuration</th>
<th>Performance of Low Capacity Cache IPC (MPKI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>equake</td>
<td>0.33 (18.3)</td>
<td>128KB, 512B</td>
<td>0.53 (2.8)</td>
</tr>
<tr>
<td>gap</td>
<td>0.23 (21.6)</td>
<td>4KB, 512B</td>
<td>0.34 (6.0)</td>
</tr>
<tr>
<td>gcc</td>
<td>0.51 (13.2)</td>
<td>4KB, 512B</td>
<td>0.55 (5.3)</td>
</tr>
<tr>
<td>gromacs</td>
<td>1.36 (1.4)</td>
<td>512KB, 256B</td>
<td>1.52 (0.5)</td>
</tr>
<tr>
<td>lbm</td>
<td>0.33 (27.3)</td>
<td>512KB, 256B</td>
<td>0.44 (9.0)</td>
</tr>
<tr>
<td>mcf</td>
<td>0.04 (141.8)</td>
<td>4KB, 512B</td>
<td>0.06 (22.6)</td>
</tr>
<tr>
<td>sphinx</td>
<td>0.69 (9.4)</td>
<td>512KB, 256B</td>
<td>0.83 (3.0)</td>
</tr>
<tr>
<td>wupwise</td>
<td>1.68 (1.1)</td>
<td>256KB, 512B</td>
<td>1.79 (0.3)</td>
</tr>
</tbody>
</table>

To dissect this behavior, we case study the benchmark gcc in detail. A sequence of LLC accesses and its hit/miss patterns for different cache configurations, labeled as <capacity, block size>, are presented in Table 3.2. From Table 3.2, it can be seen that the <1MB, 64B> baseline LLC suffers from a high miss rate for this access sequence. Although there are temporal reuses, the reuse distance is too large for the 1MB cache to capture. The 512B block size in the <4kB, 512B> LLC leverages the spatial locality and has a higher hit rate. As the access to x416201 is adjacent to the access to x416200 (only 1 cycle apart), it is merged with
the prior one using miss status handling registers (MSHRs). Therefore, we mark it with a ‘*’ in Table 3.2 to show that it is essentially a miss when the 512B block size is used. The accesses to x4161fd, x4161ff and x416202 hit in the small LLC as they are issued hundreds of cycles after the accesses to x4161fe and x416200. Note that, large blocks not only reduce cold misses, they also reduce capacity misses when the same access sequence repeats. For the <4MB, 64B> LLC, the much increased capacity is able to capture the temporal reuses.

<table>
<thead>
<tr>
<th>Block address</th>
<th>Issue cycle</th>
<th>(1MB, 64B) LLC</th>
<th>(4kB, 512B) LLC</th>
<th>(4MB, 64B) LLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>x416200</td>
<td>514782</td>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
</tr>
<tr>
<td>x416201</td>
<td>514783</td>
<td>Miss</td>
<td>Hit*</td>
<td>Miss</td>
</tr>
<tr>
<td>x4161fe</td>
<td>515077</td>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
</tr>
<tr>
<td>x4161fd</td>
<td>515374</td>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
</tr>
<tr>
<td>x4161ff</td>
<td>515806</td>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
</tr>
<tr>
<td>x416202</td>
<td>516289</td>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x416200</td>
<td>13525943</td>
<td>Miss</td>
<td>Miss</td>
<td>Hit</td>
</tr>
<tr>
<td>x416201</td>
<td>13525944</td>
<td>Miss</td>
<td>Hit*</td>
<td>Hit</td>
</tr>
<tr>
<td>x4161fe</td>
<td>13526238</td>
<td>Miss</td>
<td>Miss</td>
<td>Hit</td>
</tr>
<tr>
<td>x4161fd</td>
<td>13526535</td>
<td>Miss</td>
<td>Hit</td>
<td>Hit</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another important observation from Table 3.2 is that although the access pattern may be challenging for data prefetchers, large cache blocks can effectively exploit spatial locality. In other words, large blocks and data prefetchers are complementary in exploiting spatial locality. More results presented later (in Section 3.6) confirm our observation where many benchmarks show performance improvements even in the presence of aggressive prefetching. Here, it is noteworthy that our observation on capacity impact from spatial locality also holds for data prefetchers: a perfect prefetcher obviates the need of cache capacity and the required cache size can be arbitrarily small. Our baseline processor model includes an aggressive stride-detecting streaming data prefetcher in our experiments. A better prefetcher can
complement SLCP more by freeing up more capacity for the benchmarks with better spatial locality.

Inspired from such locality analysis, we propose an online locality monitoring framework (Section 3.3) to capture locality information at run-time. Then, in our proposed locality-driven cache management algorithms (Section 3.4), we exploit the tradeoffs between cache capacity and block sizes. In single-core processors, for applications with strong spatial locality, we can use a large cache block size to achieve high performance and reduce the capacity requirement, which enables part of the cache to be turned off. In multi-core processors, spatial locality is leveraged for a shared LLC such that the benchmarks with strong spatial locality can effectively donate extra capacity to other co-scheduled workloads. Essentially, all these are achieved by using online locality information to determine the optimal capacity and block size for different applications. Such heterogeneity in cache hierarchy introduces high performance gains and energy savings.

We make following contributions in this chapter:

a. We propose online locality monitoring, a simple mechanism to measure both spatial and temporal locality at runtime.

b. For single-core processors, we propose an algorithm, referred to as Hcache, to dynamically configure private LLC organizations.

c. For multi-core processors, we propose a spatial locality-aware cache partitioning (SLCP) scheme for a shared LLC.

d. We present detailed performance and energy evaluation to show the effectiveness of our proposed schemes. We also use case studies to show why a recently proposed partitioning algorithm PriSM [47] makes suboptimal partitioning decisions. We highlight that due to the tradeoffs between the block size and the cache capacity requirement, the optimal partitions can only be obtained when the two parameters are considered together, as in our proposed SLCP approach.

The remainder of this chapter is organized as follows. Section 3.2 presents related works and distinguishes our work from them. Section 3.3 presents the microarchitecture of our online
locality monitors. Section 3.4 presents the proposed cache reconfiguration and cache partitioning algorithms, and Section 3.5 describes the methodology of experiments. Section 3.6 discusses the experimental results. Section 3.7 summarizes our findings.

3.2. Related Work

There has been a significant amount of research work on dynamically reconfiguring cache organizations as well as cache partitioning. In [16][67], cache architectures are proposed such that their cache associativity, cache block size or cache capacity are reconfigurable. However, they do not leverage the important tradeoff between the capacity and the block size for cache management.

In the adaptive line-size cache [64], the virtual block size is adjusted based on whether there are neighboring sub-blocks in the cache. Spatial footprint prediction [42] and spatial pattern prediction [10] predict which sub-blocks in a large block are utilized and reduce the data traffic by only fetching the useful ones. They are complementary to the adaptive block sizing when memory bandwidth of the system is constrained.

For shared LLCs, both coarse-grain and fine-grain partitioning schemes have been proposed. Coarse-grain partitioning schemes partition the cache by partitioning cache ways [12][17][52][61][66], partitioning cache sets [53][67] or page coloring based partitioning [45]. Fine-grain partitioning schemes, such as PIPP [72], PriSM [47], Gradient-Based Cache Partitioning [30], and Vantage [56] can partition the cache at a fine granularity up to cache blocks. UCP [52], COOP [77], PIPP, Gradient-Based Cache Partitioning, and PriSM use temporal locality/utility based analysis to make their partitioning decisions. Vantage provides the fine-grain partitioning capability but requires caches to have good hashing as well as high associativity to work well. It has been shown that PriSM significantly outperforms Vantage for LLCs with simple indexing and moderate associativity [47]. Other than above mentioned cache partitioning proposals, there are related works on fair cache sharing [41] and quality of service (QoS) [32]. But these proposals mainly differ in the end goals of cache partitioning and can be easily realized in our online locality monitoring framework.
With the previous works [64][42][10] showing the benefits of large cache blocks in single cores, our work makes the case for large blocks by highlighting the benefits in improving cache energy efficiency and in shared cache partitioning. Our work differentiates from the previous works in that (1) we leverage the trade-off between spatial and temporal locality to more efficiently utilize cache capacity and reduce the LLC energy consumption without sacrificing overall performance whereas related works achieve energy savings at small performance losses; (2) we leverage spatial locality to reduce the capacity requirement so as to effectively share the LLC among multiple cores; (3) the locality measure used in our work focuses on improving cache hit rates rather than the utilization rate of sub-blocks in a large block. Spatial footprint or spatial pattern prediction are complementary to our approach and they can be utilized to reduce the memory traffic caused by large blocks. Recently proposed adaptive granularity memory systems [73][74] offer the capability of providing data at different granularities on the off-chip memory system side, which improves bandwidth utilization and system efficiency for small cache blocks. The techniques presented in this research are also complimentary to such adaptive granularity memory system proposals.

### 3.3. Online Locality Monitors

In this section, we elaborate on our online locality monitoring framework and present the microarchitecture of locality monitors.

As discussed in Section 2, the conditional probability based locality measure offers a unified way to quantify temporal and spatial locality. To estimate this locality measure in a dynamic fashion, we design hardware-based online locality monitors. A naïve way to do so is to maintain a fully associative structure for each capacity and block size combination. This is obviously impractical for hardware implementation. Therefore, we make the following two simplifications to make online locality monitoring feasible.

(1) Rather than a fully associative structure of tags; we use set-associative structures. The set associativity is the same as the baseline cache so as to facilitate way-partitioning used in our partitioning scheme (described in Section 3.4.3). If we assume uniform accesses across
all the sets, this method does not lead to any significant error in locality computation. The assumption holds true for most of the benchmarks in our study.

(2) The amount of tag storage required is substantial if we maintain tags for all cache blocks in the cache. Therefore we use set sampling [52] to mitigate the hardware overhead of online locality monitoring.

With these two simplifications, we propose the online locality computation logic as shown in Figure 3.2. To capture the locality information of the access stream of a cache, we introduce a few sets of auxiliary tag directories (ATD) and locality score (LScore) counters. As the locality scores for one block size cannot be used to derive the locality scores for other block sizes, multiple ATDs and LScore counters are employed. All ATDs have the same numbers of ways as in the original cache so as to capture the locality information for different cache capacity. As shown in Figure 3.2, ATD-64 is used for our baseline LLC with a 64B block size and ATD-128/ATD-256/ATD-512 maintains the tags at the 128B/256B/512B cache block granularity.

With set sampling, the ATDs monitor the address stream to the sampled sets in the baseline LLC. One important issue here is to ensure that all ATDs monitor the same address stream, which constrains the beginning index of the sampled set to be a multiple of 8 in this case. The reason is that the address stream to a set (e.g. set 1) in ATD-512 should be the same as the access stream to 8 consecutive sets (i.e. sets 8-15) in the baseline cache/ATD-64.

The LScore counters record the number of hits in the ATDs. ATD-64 updates LScore-64; ATD-128 updates LScore-128; and so on, as shown in Figure 3.2a. Taking the LScore-64 counters as an example, its first counter maintains the number of hits if the MRU position (i.e., 1 cache way or 1/assoc. of cache capacity) is assigned to a thread. Its second counter maintains the number of hits when both the MRU position and the (MRU-1) position (i.e., 2 cache ways or 2/assoc. of cache capacity) are assigned to a thread. Therefore, for each hit at the LRU stack position $\alpha$, the LScore counters at positions $\alpha$ to assoc. are incremented, where assoc. is the associativity of the cache. This way, the LScore counters provide the cache hit information when different number of ways and various block sizes are assigned.
To obtain a final/normalized locality score (or a hit rate), the LScore counter can be divided by the number of accesses to the sampled sets. For the notation purpose, a \( L_{\text{Score}}(L,K) \) refers to the \( L^{\text{th}} \) entry in the LScore-K counter and it maintains the number of cache hits for the cache block size of \( K \) and the capacity of \( L^*(C_{0}/\text{assoc.}) \), where \( C_{0} \) is the baseline cache capacity and \( (C_{0}/\text{assoc.}) \) is the capacity of one cache way. Figure 3.2b shows that for a multi-core system, there is a set of Lscore counters per core to monitor the miss stream of each core’s private L2 cache (or the per-core access stream to the shared LLC).

![Diagram of ATDs and Locality Scores (LScores) for locality computation](image)

**Figure 3.2:** Organization of (a) ATDs and Locality Scores (LScores) for locality computation (b) 2D-array of LScores per core

In our experiments, we observed that online locality monitors estimate the locality accurately even when we sample 32 sets out of 4096 sets. As a result, the on-line locality monitors have low hardware overhead. In our design, we divide the 32 sampled sets into two groups of 16 sets and the two groups were randomly placed for sampling. For a 16-way cache, we need \((32*16 + 16*16 + 8*16 + 4*16) = 960 \) ATD entries core to monitor the block sizes of 64B, 128B, 256B, and 512B. Each ATD entry is 37 bits (= up to 32-bit tag + 1 valid bit + 4 LRU bits). Therefore, the storage cost of ATDs is 4.34kB per core. We use 32-bit
LScore counters and the cost of LScore counter array is \((16\times4)\times4B = 256B\). Hence, the online locality monitor has an overall storage overhead of 4.59kB per core. For a 4-core system with 4MB LLC or a single-core system with 1MB LLC, the overhead is 0.45%.

### 3.4. Locality Driven Cache Management

Since we exploit the tradeoff between cache capacity requirements and cache block sizes, we first discuss how to realize different cache block sizes efficiently. Then, in Section 3.4.2, we present our locality-driven cache reconfiguration algorithm, called Hcache algorithm, to improve the energy efficiency for an LLC in a single-core processor or a private LLC in a multicore processor. In Section 3.4.3, we describe our Spatial Locality-aware Cache Partitioning algorithm (SLCP) for shared LLCs in multi-core processors.

#### 3.4.1. Dynamically Adjusting Cache Block Sizes

We propose to use next/previous n-line prefetching support large cache blocks. When the cache management algorithm determines that the appropriate block size is larger than the baseline block size, the cache controller tries to fetch \(n\) blocks, where \(n\) is the ratio of the large block size/the baseline block size, with the missing block prioritized. Depending on the missing address’ offset within the large block, previous lines and/or next lines are prefetched if they are not in the cache or moved to the MRU (most-recently used) positions otherwise. The granularity of replacement is unchanged and we still maintain LRU at baseline cache block size granularity (i.e. 64-byte here).

<table>
<thead>
<tr>
<th>Cache block size (byte)</th>
<th>Total area (mm^2)</th>
<th>Total read dynamic energy per read port (nJ)</th>
<th>Total standby leakage power per bank (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>10.76</td>
<td>0.84</td>
<td>0.93</td>
</tr>
<tr>
<td>128</td>
<td>15.86</td>
<td>1.50</td>
<td>0.95</td>
</tr>
<tr>
<td>256</td>
<td>29.00</td>
<td>5.58</td>
<td>1.37</td>
</tr>
<tr>
<td>512</td>
<td>79.21</td>
<td>21.07</td>
<td>2.84</td>
</tr>
</tbody>
</table>
Using next/previous n-line prefetching to emulate large block sizes is preferred to having specific physical designs for large block sizes. In Table 3.3, we report the total area, access energy and leakage power of a 16-way set associative 4MB LLC using 32nm technology for different cache block sizes, which are computed from CACTI. Clearly, using a smaller cache block size is better for the total area and leakage power considerations. From the read access energy standpoint, accessing a 128B block is slightly cheaper than accessing two 64B blocks. But with the block size of 256B and 512B, the access energy increase is significant, which makes them a poor design choice. Therefore, we choose to emulate cache block sizing instead of changing the physical design to have bigger cache block sizes.

3.4.2. Locality-Driven Cache Reconfiguration for Single-Core Processors: Hcache Algorithm

With the locality scores collected from online locality monitors, we determine the optimal \( \langle \text{cache capacity, block size} \rangle \) configuration for an application. To realize different cache capacity, we resort to power gate schemes to turn on/off individual cache ways.

Our cache reconfiguration algorithm, referred to as \textit{Hcache}, essentially picks the configuration with high locality scores and low energy cost. The algorithm is described as follows:

We start with the locality scores corresponding to different cache line sizes and capacities, which are obtained from the on-line locality monitors. For each configuration \( \langle C, B \rangle \), we first compute the number of misses for this configuration from the locality score:

\[
\text{Miss\_score}\langle C, B \rangle = \text{Total\_accesses} - L\text{score}(L,B), \quad \text{where } L = (C / C_0)^\text{assoc.} \text{ and } C_0 \text{ is baseline (total) LLC capacity.}
\]

Then, we convert the number of misses of a cache configuration into a cost value by considering the bandwidth cost from large block sizes and the energy cost from a larger cache capacity. In other words, we want to ensure that we only choose a large block size or a large capacity when there are significant reductions on cache misses. The equation for such conversion is as follows:
\[
\text{Cost}_{C, B} = (1 + T_b \cdot \log_2\left(\frac{B}{\text{Baseline block size}}\right)) \times (1 + T_c \cdot \log_2\left(\frac{C}{\text{Minimum capacity}}\right)) \times \text{Miss\_score}_{C, B},
\]

where \(T_b\) and \(T_c\) are knobs to adjust how much additional cost we charge to increased block sizes and enlarged capacity, respectively, and the minimum capacity represents the capacity of the smallest cache configuration available for the cache (e.g., 1 cache way).

The Hcache algorithm then picks the configuration with lowest cost with an exception: when DRAM bandwidth is saturated (i.e. the observed memory bandwidth consumption > \(BW\_threshold\)), it chooses to fall back to the baseline cache configuration. In our experiments, we determine the value of the controllable knobs, \(T_b\), \(T_c\) and \(BW\_threshold\) empirically to optimize performance.

### 3.4.3. Spatial Locality-aware Cache Partitioning for Shared LLC in Multicores

In this section, we elaborate on our spatial locality aware cache partitioning algorithm for effective cache sharing in shared LLCs. As we mentioned before, with large cache block sizes, applications can achieve similar or better cache hit rates with reduced capacity. This enables these benchmarks to donate more LLC capacity to others benchmarks compared to when they are using the baseline cache block size. In other words, the benchmarks with strong spatial locality become effective donors of capacity while the benchmarks favoring temporal locality utilize the extra cache capacity to improve their performance.

The goal of our partitioning algorithm is to decide both the capacity and the block size for each core so as to achieve the best overall performance. Therefore, this is a two-dimensional optimization problem rather than a one-dimensional optimization, as in the previous works such as Utility-based Cache Partitioning (UCP) [52] and Probabilistic Shared Cache Management (PriSM) [47]. To achieve our goal, we use our online locality monitors to measure the unified spatial and temporal locality at runtime. The locality information is then used to determine the optimal heterogeneous organization for each core under the constraint of the overall LLC capacity.
Figure 3.3 shows a high level organization of processors and cache hierarchy in multicore architecture. We add two hardware structures in the LLC controller. The first is the locality monitors (i.e., ATDs and Lscore counters). The second is the partitioning algorithm, which creates one partition per core in the LLC and assigns the capacity and block size to each partition.

Realizing such heterogeneous cache partitions requires little hardware change to the regular set-associative cache structure. We leverage the previous works [52][61] on modifying the replacement algorithm to partition a cache. As described in [52], each cache block maintains a thread-id to specify which thread brought the block into the cache. Upon a miss, the replacement algorithm first determines if the allocated blocks to the missing thread are lesser than assigned partition size. If yes, the LRU block among other threads’ blocks is chosen as a victim, otherwise the LRU block of the missing thread will be replaced.

In a cache-partitioning problem, the optimal partitions maximize/minimize a particular performance metric. In this work, we choose to maximize the weighted sum of locality scores across all the co-scheduled benchmarks. The benefit for such a metric is that as the locality scores are indicative of hit rates and maximizing hit rates improves throughput. As our per-
core LScore counters provide cache hit rates for different capacity and block size combinations, the goal of our partitioning algorithm is to choose a point from each 2-D array of LScore counters so that the weighted sum of locality scores is maximized under the condition of overall cache capacity. The weights here are the LLC access rate of each core. As a result, the weight sum effectively obviates the need of normalizing the locality scores, i.e., dividing the LScore counters by the number of accesses to each core. We now discuss how we obtain the capacity and the cache block size for each partition.

To solve this optimization problem, one possible solution is an exhaustive search, in which we pick one point from each LScore counter array and try all the possible combinations. This approach incurs too much overhead. Instead, in our approach, we first reduce a two-dimensional Lscore array to a single-dimension Lscore array and then leverage the lookahead algorithm to find the partitions as described in [52].

As an Lscore array maintains the number of hits for different block sizes with each possible capacity assignment, it is possible to first pick the best block size for each capacity assignment. This way, a two-dimensional Lscore array for each core is reduced to single-dimensional Lscore vector with each Lscore annotated with its associated block size. Considering extra memory traffic incurred by large blocks, we also employ a weight function to scale the single-dimensional Lscore vectors. For a cache block size $K$, the weight function $W(K)$ is defined as $1 - \frac{\log_2(K) - \log_2(K0)}{\beta}$, where $K0$ is the baseline block size and $\beta$ is a design parameter. The parameter $\beta$ adjusts the improvement in locality scores based on block sizes. For example, comparing $\beta = 32$ vs. $\beta = 16$, the weight function, $W(128)$ is $31/32$ vs. $15/16$ for a baseline block size of 64B. In comparison, $W(256)$ becomes $30/32$ vs. $14/16$. The purpose of such scaling is to increase the block size only when the locality score improvement is significantly high.

As the last step of our SLCP algorithm, the lookahead algorithm [52] is used upon the scaled single-dimensional Lscore vectors, one vector per core, to determine the capacity for each partition. The lookahead algorithm starts with an assumption that each thread will be
assigned at the least one way in the cache. Further capacity assignment is based on increment in locality per unit capacity (analogous to the marginal utility used in UCP algorithm). The core with the maximal increment in locality per unit capacity is assigned the required capacity. After an iteration of capacity assignment, at least one cache way is assigned to one of the cores/threads. This ensures that partitioning algorithm terminates in at most (set associativity - number of cores) steps. Then, the block size, annotated with the chosen capacity for a partition, is used to determine the granularity of cache operations on this partition.

As an example, for a 16-way set-associative 1MB cache with block size of 64B, we may get following partition configuration as the output of the lookahead algorithm: Core0 (6, 64); Core1 (3, 128); Core2 (5, 64); and Core3 (2, 256). To realize the such partition configuration, SLCP assigns 6 ways to the first core, 3 ways to the second core with next/previous line prefetching enabled to fetch 128-bytes of data blocks, 5 ways to the third core, and 2-ways to the fourth core with next/previous 4 line prefetch enabled to fetch 256-byte data blocks.

The cache partitioning algorithm is invoked to determine partition configurations after a fixed amount of time/cycles (e.g. every 1M cycles or 5M cycles). The purpose is to let the LScore counters to accumulate some representative locality data over time. Here, a key assumption is that the currently observed program phase stays similar to the next. We observe that the locality score profile does not change significantly during such a time interval. Therefore the partitions determined based on recent information tend to work well. To prioritize the recent behavior over the past behavior without losing the history completely, we right shift all the LScore and access counters once, rather than resetting them, after new partitions are created. This is a simple and effective operation used in related works such as [52].

3.5. Experimental Methodology

We conduct our experiments using an in-house execution-driven simulator. It uses the SimpleScalar [7] frontend and its timing model simulates a MIPS R10K style out-of-order
(OOO) execution pipeline. In our multicore system, each core is a 4-way issue superscalar processor with a 64-entry active list.

The memory hierarchy has three levels of caches which include L1 and L2 caches private to each core and a shared L3 cache (non-inclusive). We simulate a 32kB L1 data (D) and instruction (I) cache with the block size of 64 bytes and 1-cycle hit latency. The L1 D- and I-caches are 4-way and 2-way set associative, respectively. The L2 caches are 256kB 8-way set associative with 64 byte blocks (10 cycle hit latency). Both L1 and L2 caches use 1-bit NRU replacement policy (LRU is approximated by 1-bit NRU commonly in modern high performance processors [34]) Single core system and 4-core system have 1MB and 4MB L3 cache respectively with the block size of 64 bytes (30 cycle hit latency). Our baseline system has no partitioning scheme deployed for LLC and it uses the LRU replacement policy. The baseline also employs a stride-based stream buffer prefetcher [56] for each core. The prefetcher observes the L1 miss-stream and prefetched the data in the L2 cache and the L3 cache. Each prefetcher has 16 eight-entry stream buffers with a PC-based 4-way 1024-entry stride prediction table. When detecting a confident stride, the prefetcher can request up to 4 cache blocks in L2 cache if the L2 MHSRs are available (i.e. prefetch degree of 4). We find that beyond prefetch degree of 4 the overall performance improvement of the system is saturated.

We model a detailed main memory system and off-chip bandwidth using the DRAMsim2 [55] framework. We model a DDR3-1600 system with one rank and 8 banks (per bank row buffer size = 4kB). The DRAM transactions are handled on the 64-byte granularity and an open-row policy is used with higher priority for row-hits. For single-core system, we have one channel of 6.4GBps off-chip bandwidth and our 4-core system has 2 channels, which provide shared 12.8GBps bandwidth among 4 cores. Lower bits of the cache block address are used for mapping contiguous blocks across different channels for higher parallelism.

We use SPEC2000 and SPEC2006 benchmark suite for our experimental evaluation. We include all the benchmarks that we were able to compile and run successfully using the SimpleScalar ISA. We have included 16 benchmarks from SPEC2000 and 8 benchmarks
from SPEC2006 in this study. All these benchmarks are run with their reference input and we use 100M simulation points determined using SimPoint [29]. The multiprogrammed workloads are created by mixing these program phases and the simulation statistics are collected for the representative phase only, though we maintain contention in shared cache by keeping the other benchmarks running beyond their simulation phase until all the benchmarks in the multiprogrammed workload have finished their SimPoint phases.

To create 4-way multi-programmed workloads, we categorize the benchmarks into two groups: high MPKI (Misses Per Kilo Instructions) benchmarks and low MPKI benchmarks. Among 24 benchmarks, 9 are found to be high MPKI (MPKI > 5) and other 15 fall in the category of low MPKI benchmarks. Note that we mix the benchmarks based on their memory intensive behavior. We do not distinguish them based on spatial locality behavior and maintain fairness in creating representative multi-programmed workloads.

We create following five categories of 4-way multi-programmed workloads: (a) 4H: all 4 benchmarks with high MPKI; (b) 3H1L: 3 benchmarks with high MPKI and 1 benchmark with low MPKI; (c) 2H2L: 2 benchmarks with high MPKI and 2 benchmarks with low MPKI; and (d) 1H3L: 1 benchmark with high MPKI and 3 benchmarks with low MPKI; and (e) 4L: 4 benchmarks with low MPKI. In each of these categories, eight mixes are created by randomly picking benchmarks (i.e. a total of 40 workloads). Similar methodology is adapted for creating 8-way multiprogrammed workloads (i.e. categories 8H, 6H2L, 4H4L, 2H6L).

In the results section (Section 3.6), we compare the performance of the baseline, our SLCP approach and related works using instruction per cycle (IPC) throughput and weighted speedup (WS) [20] metric for comparing throughput of the system.

\[
IPCThroughput = \sum IPC_{shared}[i]
\]

\[
WS = \sum (IPC_{shared}[i]/IPC_{alone}[i])
\]
3.6. Experimental Results


In this section, we show energy savings obtained by the Hcache algorithm for the LLC in a single-core processor (or the private LLC in a multi-core processor). As mentioned in Section 3.4.2, the impact of Tb and Tc parameters in the Hcache algorithm is explored in our experiments. We compare Hcache with a capacity_only scheme where the goal of optimization is to reduce cache energy consumption within the constraint that the application must have a less than 3% slowdown. The capacity_only scheme is similar to related works, such as [1], where small amount of performance is traded-off for significant savings in cache energy consumption (i.e. energy savings by turning off major portion of the cache when extra cache capacity has negligible gains). In this capacity_only scheme, we resort to an oracle study, which exhaustively search the optimal LLC configurations for each application. This provides us with a reasonable estimate of the upper bound of the EDP reduction and energy savings when a reconfiguration algorithm is applied focusing only capacity. However, it does not exploit the tradeoff between capacity and block sizes.

Figure 3.4 shows the results of the H-cache when \( T_c = 0.2, T_b = 0.1 \) and \( BW_{\text{threshold}} = 6.0 \) GB/s. The minimum capacity configuration has 1/8 capacity of total LLC size (i.e. 2 cache ways for 16-way set associative LLC) and capacity reconfigurations are chosen to be power of 2 multiple of minimum capacity. The results of capacity_only scheme are also included for comparison. From Figure 3.4, we can observe that the benchmarks such as equake and sphinx show significant performance improvement when bigger cache blocks are used (as also shown in Table 1) by Hcache. Due to limited temporal locality for baseline block sizing, the capacity_only scheme decided to turn off major portion of the capacity and leads to high L3 energy savings though it shows slight slowdown in performance. On the other hand, our Hcache algorithm chooses to operate at lower capacity but bigger block sizing (to exploit spatial locality), thereby saving cache energy and increasing performance at the same time. Benchmarks like milc and perl do not have good spatial locality and both the
Hcache and capacity_only scheme turns off major portion of the cache to save energy and drops the performance slightly.

Overall across our 24 benchmark applications, Hcache can save 34.9% L3 energy consumption on average using the geometric mean (Gmean) while decreasing the average EDP by 10.7% and increasing performance by 5.6%. On the other hand, the capacity_only scheme saves 35.9% L3 energy consumption but increases EDP by 0.6% and shows 0.9% loss in performance. In conclusion, Hcache is able to leverage the tradeoff between capacity and block sizes, and shows similar L3 energy savings to the capacity_only scheme, but it also improves performance and system EDP.

### 3.6.2. Evaluating SLCP for Shared LLCs in Multi-core Processors

In this section, we first show the results of a naïve combination of leveraging spatial locality and cache partitioning, which highlights the importance of spatial locality in cache partitioning decisions. Then, we compare results of SLCP with related work and discuss why
it makes better partitioning decisions. Sensitivity study to algorithm parameter and energy savings of SLCP are also discussed in this section.

3.6.2.1. Evaluating SLCP for Shared LLCs in Multi-core Processors

We evaluate the performance of a 4-core system where both techniques to exploit spatial locality and cache partitioning are employed. We use UCP as the cache partitioning scheme here. To exploit spatial locality, we statically change the cache block size for all the benchmarks or use adaptive line sizing (ALS) proposed by Veidenbaum et al. [64]. Our ALS implementation chooses the block sizes from 64B up to 512B in a dynamic fashion. We tuned ALS to generate the best performance with the increase-threshold as 0.5 and the decrease threshold as 0.7. Note that the baseline processor already has an aggressive stride-based data prefetcher.

Compared to a design without any data prefetchers, the baseline system shows high improvement in weighted speedup with a geometric mean of 21.2%, confirming the importance of spatial locality. Figure 3.5 shows the comparison of weighted speedup (WS) of the system with UCP, UCP with different cache block sizes at LLC and ALS. We only show geometric mean of normalized WS of each category due to space constraint (a total of 40 benchmarks, 8 in each category as mentioned in Section 3.5). Compared to the baseline, UCP achieves 6.9% performance improvement by applying cache partitioning. When UCP is used in conjunction with bigger block sizes such as 128-byte, 256-byte or 512-byte, it leads to significant performance improvements showcasing the importance of exploiting spatial locality in combination with cache partitioning. We also observe that UCP combined with a 256-byte block size is the best performing static design on average but for category 1H3L a block size of 512B performs the best. The reason is due to less bandwidth consumption in the baseline, increasing the cache block size to 512-byte does not hurt performance while for other benchmark categories a 512B block size saturates the off-chip bandwidth and 256-byte blocks perform better. The 4L category has low bandwidth as well but it has all low MPKI benchmarks and therefore the performance impact of block sizing is limited.
From Figure 3.5, we can see that, on average, ALS outperforms UCP with 64-byte block size and 512-byte block size but other UCP configurations outperform ALS. Therefore, UCP’s performance is significantly affected by the choice of cache block size. ALS, on the other hand, only uses dynamic block sizing and can outperform UCP (and perform similar to UCP with 128B blocks) even in absence of cache partitioning. ALS can detect spatial locality dynamically and hence reduces capacity requirement for some applications within many multiprogrammed workloads. This reduces the contention for cache capacity and hence achieves good performance even in the absence of partitioning. This confirms the significant impact of exploiting spatial locality for shared LLCs in multi-core processors.

### 3.6.2.2. Performance Improvements of SLCP

In this section, we compare SLCP with other cache partitioning schemes UCP, PriSM and adaptive line sizing (ALS). For our 4-way multiprogrammed workloads, Figure 3.6a and 3.6b show the normalized IPC throughput and weighted speedup (WS), respectively when different schemes are applied to L3 cache. For PriSM, we use the PriSM-H component of the proposal because it focuses on maximizing hits and hence the system throughput. For brevity, we refer to PriSM-H as PriSM. We implement both UCP and PRiSM in our simulation framework. In our implementation, UCP uses the lookahead algorithm to search for the sizing of cache partitions. Both UCP and PRiSM deploy 1 sampled set per 32 cache sets for dynamic set sampling. All other algorithm specific parameters of these two algorithms are used as described in the previous works. SLCP and ALS explore the same range of block sizes (i.e. 64B/128B/256B/512B) to provide a fair comparison. We use $\beta = 8$
in the lookahead algorithm and the partitions are reconfigured every 5M cycles. We also assume that the online locality monitoring hardware and cache partitioning scheme does not affect the cycle/access time of LLC. This assumption is valid here because our locality analysis is off the critical path of cache accesses.

Figure 3.6. Normalized IPC throughput and weighted speedup of UCP, ALS, PriSM and SLCP for 4-way multiprogrammed workloads.

From our simulation results shown in Figure 3.6, ALS shows IPC throughput (weighted speedup) improvement of 8.1% (9.1%) on average by reducing the capacity requirements of benchmarks which in turn reduces contention for capacity and improves performance. UCP and PriSM aim at partitioning the capacity to maximize LLC hits but only considers temporal locality while making partitioning decisions. Therefore, UCP can only achieve 7.2% (6.9%) of improvement while PriSM does not improve performance overall. On the other hand, SLCP leverages spatial locality to determine the appropriate configuration of each partition and shows 18.2% (20.9%) performance improvement. This highlights the importance of leveraging spatial locality in a managed way. The performance improvements from SLCP are due to two key factors. First, SLCP is spatial locality aware and assigns less capacity to the benchmarks which require lesser cache capacity when spatial locality is exploited. The
capacity partitioning capacity prevents the prefetched blocks from interfering with co-scheduled applications. Second, as mentioned in Section 3.1, for benchmarks whose access behavior has strong spatial locality but hard to detect stride patterns, large blocks are an effective way to complement data prefetchers.

From Figure 3.6, it can be observed that the multiprogrammed workloads with a higher number of high MPKI benchmarks tend to improve more with SLCP (e.g., 4H or 3H1L). There are two main reasons. The first is that high MPKI benchmarks compete for cache capacity and they tend to thrash others’ data, resulting in lower performance in the baseline. The second is that many high MPKI benchmarks show good spatial locality (e.g., those listed in Table 1) and their working sets can be significantly reduced with the use of large cache block sizes. Therefore, these workloads see significantly better hit rates as well as IPC improvements while at the same time others benefit from extra capacity donated by those benchmarks.

For an 8-core system, Figure 3.7 shows the comparison of weighted speedup improvement of SLCP compared with that of UCP, ALS and PriSM. On average, UCP achieves good performance improvements of 9.6% while impact of ALS on performance is limited in this case (i.e. 3.2% improvement). Similar to 4-core results, PriSM does not show performance benefit for most of the workloads and suffers an average throughput degradation of over 18%. Unlike 4-core results, in 8-core results UCP performs better than ALS. The reason for that is that the both systems have a 2-channel DRAM memory system. In this manner, 8-core system is much more bandwidth restricted compared to 4-core system and has less room for aggressive block sizing from ALS. In this scenario, optimizing on temporal locality using UCP is more helpful due to bandwidth constraints. This indicates that with more programs sharing an LLC, capacity partitioning becomes more important not only for isolation but also for better performance. Nevertheless, SLCP shows significantly better performance than both techniques and achieves 18.4% improvement in IPC throughput. In the following section we address if SLCP’s spatial locality aware partitioning has merit over
simply combining the cache partitioning (e.g. UCP) and adaptive cache block sizing (e.g. ALS).

3.6.2.3. *Is Exploiting Spatial Locality orthogonal to Cache Partitioning?*

In this section, we address a key question that if including spatial locality into cache partitioning problem has any merit? Or in other words, is joint optimization of spatial locality and cache partitioning can be significantly better than optimizing them independently. In the results discussed previously, UCP does not employ any dynamic block sizing scheme and rely on LLC prefetchers to exploit the spatial locality present in the benchmarks. In this experiment, we deploy ALS in conjunction with UCP for comparison.

In Figure 3.8a, we show the geometric mean of improvement in IPC throughput for each workload category as well as geometric mean across all 40 multiprogrammed benchmarks. UCP and ALS improve IPC throughput by 7.2% and 8.1% respectively. We observe that the combination of UCP and ALS is able to achieve more performance than either of them in most cases, indicating some potential in combining the two techniques. However, for the 1H3L category, the combination of UCP and ALS hurts the performance (both throughput and weighted speedup). On average, the combination of UCP and ALS performs 8.9% better than the baseline which is significantly less than what a combination of these two techniques should achieve if they were orthogonal optimizations (i.e. $1.072 \times 1.081 \approx 1.159$ or 15.9%.
improvement). It is very interesting to see now that our proposed SLCP partitioning algorithm, which considers cache block sizing and capacity in a unified fashion for cache partitioning, achieves the highest performance in all categories and on average provides 18.2% better throughput. In this manner, we show that considering the two optimizations in a joint fashion can enable the full performance potential to be extracted while it is not possible by considering them independently. Overall, SLCP itself is 8.5% better than applying UCP and ALS both to the system. The fundamental reason is that ALS applies line sizing without considering its impact on the capacity while UCP applies partitioning without taking adaptive line sizing into account. With independent operation of ALS and UCP, the capacity assignment and block sizing decisions are suboptimal, as shown in detail in our case study in Section 3.6.2.4.

Figure 3.8: (a) Normalized IPC throughput for UCP, ALS, UCP+ALS, and SLCP over the baseline, (b) Normalized weighted speedup of UCP, ALS, UCP+ALS and SLCP.

Similar observations can be made from results based on weighted speedup metric (Figure 3.8b), where UCP and ALS achieve 6.9% and 9.1% improvement in weighted speedup
respectively whereas UCP+ALS only shows 8.4% improvement (i.e. slowdown from combining the two techniques compared to using ALS alone). This strengthens our argument that adaptive block sizing and cache partitioning are not orthogonal optimizations, and also SLCP is able to achieve 20.9% better weighted speedup in this scenario reaching the full potential of the two techniques.

3.6.2.4. Understanding Partitioning Decisions of UCP, PriSM and SLCP

We further investigate into why PriSM fails to improve performance for many of our 4-core and 8-core benchmarks while UCP and SLCP both achieve good speedups across many benchmarks we studied. We illustrate the fundamental shortcoming of PriSM in the Figure 3.9 using the case study on the benchmark mcf. The figure shows the cache hit rate when the number of ways allocated to benchmark mcf is varied from 1 to 16 in a 4MB 16-way set-associative LLC with a 64B block size. Mcf exhibits thrashing behavior as the hit rate is extremely low until the number of allocated ways is increased to 7. This curve is used successfully by UCP to assign low amount of capacity to mcf when other benchmarks are contending for capacity heavily. In this manner, UCP correctly estimates that there is practically no gain of assigning more than 1 way in the set (up to 6) to benchmark mcf. In the scenarios when UCP observes that other benchmarks are either less memory intensive or do not gain hits by extra capacity it may decide to give mcf almost half of the capacity because that is overall the best for LLC performance. PriSM’s hit maximization scheme, however, compares the difference in the hit rate of shared execution vs. running alone and assumes a linear relationship between number of ways allocated and the hit ratio (shown by the line labeled PriSM in the Figure 3.9). This is clearly inaccurate for mcf and many other benchmarks which feature similar thrashing behavior (or high temporal reuse distance). As a result, PriSM makes incorrect partitioning decisions for those benchmarks (assign capacity to application where no cache hits can be gained), leading to ineffective partitioning and poor performance as shown in Section 3.6.2.2.
On the other hand, SLCP solves the problem by adding a new dimension to the optimization problem. As shown in Figure 3.9, SLCP-128 and SLCP-256 show significantly higher cache hit rates with smaller number of ways allocated to mcf when the block size is increased to 128 bytes and 256 bytes. In other words, the relationship between hit rates and allocated cache ways is highly dependent on the block size. SLCP is designed based on this fundamental observation and it assigns 2 ways with either a 128-byte or 256-byte block size in the partition for mcf to achieve significantly better hit rate. As discussed earlier in Section 3.6.2.3, the performance of a cache partitioning algorithm like UCP combined with ALS results in low impact on performance improvement. This can be again explained from Figure 3.9. As discussed earlier, UCP monitors the access stream and decides to assign the minimum possible capacity to mcf (1 way in this case). On the other hand, by assigning just one extra way, bigger block sizing can significantly improve the hit ratio of the benchmark. With a single way allocated, large block size is not so much effective. Therefore, UCP+ALS cannot find the optimal configuration for the partition for mcf. In other words, due to the relationship between the block size and the capacity, the optimal (capacity, block size) combination can only be determined by considering capacity and block sizing together, as in our proposed SLCP approach.
3.6.2.5. Sensitivity to the $\beta$-parameter

In this section, we evaluate the impact of varying tuning parameter $\beta$ in SLCP algorithm. In Figure 3.10, we show the variation of overall performance gains of SLCP with different values of $\beta$ parameter. As mentioned before a lower value of $\beta$ parameter implies more conservative cache block sizing while a higher value will make SLCP choose bigger block sizes for marginal gains. As we model our 4-core system with a constrained DRAM bandwidth of 12.8GB/s, the overall performance improvements of SLCP increase till $\beta$ is increased up to 16. Further increase in parameter $\beta$, starts to cause performance drop due to high queuing delays at memory controller and DRAM system. This indicates that it is important to choose a suitable $\beta$ parameter value in a system with limited DRAM bandwidth.

3.6.2.6. Energy Savings of SLCP

We model the power of the multi-core processor using McPAT [44] and DRAMsim2 is used for the power model of DRAM system. We use the power numbers to calculate the overall energy consumption of the system executing the multi-programmed workloads. As different benchmarks finish at different times, we calculate the energy consumption of the cores and L2 caches for only the execution of region of interest (100M instructions each). On the other hand, we calculate the energy consumption for the shared LLC and memory subsystem till the end of the slowest application in the workload. We estimate that SLCP
requires less than 20KB of hardware overhead for a 4-core system with 4MB L3 cache. We conservatively model power consumption of the additional hardware required for SLCP as a 1% overhead over L3 cache power consumption. The relative power and energy consumption trends are shown in Figure 3.11.

First, DRAMsim2 shows that for most benchmarks the power consumption of DRAM is affected by a small fraction. This result is shown by DRAM Power bar in Figure 3.11. Even when we are increasing the off-chip bandwidth for some workloads by prefetching multiple blocks and we expect increased DRAM power consumption, overall impact on DRAM Power is relatively small. The observations from DRAMsim2 are justified as follows: accessing a bigger chunk of data while the row-buffer is open is preferred over opening the row multiple times. In multiprogrammed workloads row buffer hit rates are low due to interference of memory access streams. In essence, SLCP combines multiple distant data accesses into a burst of memory accesses and this offsets the potential increase in power consumption of DRAM.

Second, for the 32nm technology, McPAT estimates that SLCP increases the core dynamic power consumption by 12.9% compared to the baseline system as the execution time is significantly decreased. This result is shown by Core Power (core) bars in Figure 3.11. For memory-intensive workloads, such as used in our study, the sub-threshold leakage and gate leakage account for a large portion of energy consumption due to the high number idle CPU cycles. Therefore, the execution time reduction in the workloads leads to significant

![Figure 3.11: Relative power and energy consumption of SLCP with respect to baseline](image-url)
savings of 14.3% in static energy consumption of the system (DRAM + multicore). Also, our results show a 6.9% dynamic energy saving. Overall, the system energy consumption is reduced by 12.6% on average, compared to the baseline 4-core system. In comparison, UCP only provides a total saving in energy consumption of 6.9% on average and PriSM increases energy consumption because of increased execution time for the workloads.

### 3.6.3. Evaluating Multi-level Cache Optimizations

In this section, we evaluate the impact of applying spatial locality optimizations for L2 cache in conjunction with our proposed spatial locality aware cache partitioning. We choose ALS scheme for L2 cache to exploit spatial locality, and L3 cache is applied with UCP, ALS and SLCP in different experiments. The algorithm parameters as kept same as the experiments in previous sections. Figure 3.12 shows the results of such combined optimizations. For comparison, the first bar for each workload category shows the performance improvement of SLCP at L3 cache when baseline L2 cache is used.

![Figure 3.12: Normalized IPC throughput of UCP, ALS, UCP+ALS and SLCP at L3 cache when ALS is applied at L2 cache](image)

It is interesting to see that though ALS and UCP combined at L3 cache were not able to performance significantly when used in conjunction (recall Figure 3.8), ALS at L2 cache combined with UCP partitioning algorithm at L3 cache shows significant performance
improvement of 19.6%. It should also be noted that when ALS is used at both L2 and L3 cache it does not lead to significant performance improvement as they are not orthogonal optimizations and fetch same set of cache blocks when using bigger cache line sizes.

In presence of ALS at L2 cache, ALS+UCP policy at L3 cache achieves 10.4% performance improvement is while UCP at L3 cache achieves 19.6% improvement. This result further confirms that ALS and UCP combined together at L3 cache do not assist each other in improving cache performance; instead ALS L2 + (UCP+ALS) L3 diminishes the performance compared to ALS L2 + UCP L3. In summary, independent operation of UCP and ALS for L3 cache is sub-optimal while spreading the two optimizations (i.e. ALS and UCP/SLCP) across different cache levels leads to better system performance. Moreover, when SLCP is applied at L3 cache in conjunction with ALS at L2 cache, it achieves 22.5% performance improvement. Therefore, we can conclude that either in the presence or absence of spatial locality optimization at L2 cache, SLCP is superior choice for cache partitioning scheme.

### 3.7. Summary

In this chapter, we propose online locality monitors to drive cache management. Our algorithms are based on the following key observation: exploiting spatial locality can drastically reduce the cache capacity required to sustain same or better performance for many benchmarks.

In single-core processors, we propose Hcache algorithm for private caches to reduce cache energy consumption and improve system performance, thereby reducing the system energy EDP. For multi-core processors, we propose a spatial locality aware cache partitioning (SLCP) algorithm. Our online locality monitors drive both the capacity and the block size assignment for each cache partition. In our experimental results, our proposed design of spatial locality aware cache partitioning (SLCP) algorithm outperforms high performing cache partitioning algorithms including UCP and PriSM significantly in both 4-core and 8-core systems even when these schemes are combined with an adaptive line sizing scheme. We also demonstrate that jointly optimizing spatial locality and cache partitioning can be
significantly better compared to independent operation. Our results as well as case studies indicate that SLCP makes better partitioning decisions and achieves higher performance. Moreover, we show the SLCP can also reduce the overall energy consumption of the system, making it a suitable design choice for both performance and energy consumption standpoints.
Chapter 4

Adaptive Cache Bypassing for Inclusive Last Level Caches

4.1. Introduction

With increasing working sets of applications, the performance of last level caches (LLCs) is critical to the overall computer system performance. Cache management contains two key components: (1) a replacement policy, which decides the victim block if a block needs to be replaced, and (2) an allocation policy which decides whether an incoming block should be allocated in the cache. A good cache replacement policy improves cache performance by selecting the least likely to be reused block as the victim and has been studied extensively [2][9][18][34][39][40][63][70]. A good cache allocation policy chooses to bypass a block to upper levels if it is predicted to be less useful than the blocks currently in the cache [22].

Another key design decision in cache hierarchies is the inclusion property between an LLC and upper level caches. Inclusion simplifies the hardware to support cache coherence. It enables the LLC to act like a snoop filter because a data block is guaranteed to be absent in upper levels if not found in the LLC. As a result, inclusive caches have been widely used. With inclusive caches, the allocation policy is reduced to allocate all incoming requests by
default in an LLC. This is the reason why previous bypassing algorithms [11][21][22][35][38][54][63][71] only work with non-inclusive/exclusive LLCs.

Figure 4.1 shows various flavors of memory hierarchy organization possible with strict/flexible allocation policies combined with inclusive/non-inclusive LLCs. Figure 4.1a shows a non-inclusive cache where all the incoming cache blocks from memory are allocated in all three levels of caches. The LLC is non-inclusive and therefore the evictions from LLC are silent i.e. they do not try to invalidate the evicted data blocks from upper levels. On the other hand, an inclusive LLC (shown in Figure 4.1b) will force an eviction of the corresponding data block(s) from L1 and L2 cache when a cache block is evicted from LLC. This event is also referred as back invalidation. Applying a selective allocation policy/bypassing is straight forward on a non-inclusive LLC because the selected incoming blocks from memory can be filled into L1 cache and L2 cache only (as shown in Figure 4.1c) and it does not violate the non-inclusion property. The inclusive LLC is strict about filling each incoming block from memory. This causes the inclusive cache hierarchy to be incapable of using cache bypassing or any selective allocation policy.

In this work, we propose a solution to enabling cache bypassing for inclusive LLCs. We introduce a new structure in an LLC, called a bypass buffer (BB), which keeps bypassed blocks to support the inclusion property (as shown in Figure 4.1d). Therefore, the last level cache hierarchy consists of an LLC and a bypass buffer. The bypass buffer keeps tags of the data blocks which are predicted to be less important than data present in the LLC. In this manner, the working set present in the LLC is not evicted to make room for less useful data. When a block is evicted from the LLC or BB, it invalidates the data copies present in upper level caches to maintain inclusion property. Our insight is that with a good bypassing algorithm, bypassed blocks should have a short lifetime in upper level caches. Therefore, a small BB is sufficient to ascertain that when a block is evicted from the BB, it is highly likely that its data copies in L1/L2 caches are either dead or already evicted. Furthermore, we show that our proposed BB provides an efficient way to collect the usage information of bypassed blocks, which can be used to simplify and facilitate the design of bypassing algorithms.
Figure 4.1: Memory hierarchy organization for (a) a non-inclusive LLC (b) an inclusive LLC (c) a non-inclusive LLC with bypass (selective fill of L3 cache) (d) an inclusive LLC with a bypass buffer to support cache bypassing.
The key contributions of this chapter include:

1. We make an important observation on the lifetime of bypassed blocks to motivate our low overhead BB idea.

2. We show that our proposed BB facilitates the design of bypassing algorithms and it significantly reduces the hardware cost of the DSB algorithm [21], a top performing cache-bypassing algorithm.

3. We evaluate our proposed solution and show that our bypass-enabled LLC achieves up to 42.0% and an average of 9.4% performance improvement over an inclusive 2MB LLC with the least-recently-used (LRU) replacement policy. Compared to a recently proposed high performing replacement policy, DRRIP [34], our proposed approach outperforms it for both single-core systems, by up to 11.3% and 2.5% on average, and 4-core systems, by up to 14.0% and 1.3% on average.

4. We evaluate the performance of inclusive LLC with bypass-buffer in various cache configurations and different scenarios to conclude that bypass buffer can provide robust and effective solution to employing cache bypassing algorithms to inclusive LLCs.

The remainder of the chapter is organized as follows. Section 4.2 motivates our approach and dissects the lifetime of bypassed blocks to motivate our low cost solution to enabling cache bypassing for inclusive LLCs. Section 4.3 details our design of adaptive bypassing for inclusive LLCs. Section 4.4 presents the experimental methodology and Section 4.5 discusses the experimental results. We discuss related work in Section 4.6 and Section 4.7 concludes the chapter with a summary of findings.

4.2. Motivation

Bypassing has been shown to be high performing by previous research. In particular, two of the three top performers in the 1st JILP Cache Replacement Competition [80] use cache bypassing. On the other hand, many industrial designs, including Intel Core i7 [83], use inclusive last level caches which makes employing the cache bypassing on these designs a non-trivial task. As suggested in a recent work on cache bypassing algorithm [43], a
bypassing algorithm can be modified to work with inclusive caches by inserting the bypassed block at the LRU (least recently used) position. In this manner, the bypassing candidates chosen by the bypassing algorithm are victimized on the next miss to the cache set. There are two main drawbacks to this approach. First, the cache blocks still need to be placed in the cache set thereby replacing one potentially more useful block. This problem is more likely to manifest itself in a cache where the set associativity is relatively low. Second, this approach is vulnerable to a pathological scenario where many consecutive accesses are mapped to a cache set. Due to the prediction of no future reuse, they will compete for the LRU position. As a result, the lifetime of these blocks is short, which causes the victimization of the same data blocks from upper levels. This will degrade performance of inclusive LLCs. These potential performance hazards are inherently present in any cache replacement algorithm and we show in Section 4.5.4 that a benchmark (sphinx) in our experiments indeed severely suffers from this problem. Therefore, we propose to combine the bypassing algorithm with inclusive caches without converting it to a replacement algorithm. The key reason is that a bypassing algorithm is higher performing than a replacement algorithm since it does not have to insert the data in a cache level if there is no future reuse at that level of cache. Now we present our motivation behind our bypass buffer idea.

We first make an important observation on cache bypassing algorithms. The goal of cache bypassing is to bypass blocks that have fewer reuses than those currently in the cache. Therefore, for a well-designed bypassing algorithm, a block, which is bypassed from the LLC and allocated in the upper levels of caches (i.e., L1/L2 caches), should not be re-accessed after it is replaced from the L1/L2 caches. Otherwise, such re-accesses would become reuses of the block in the LLC, conflicting with the choice of bypassing. To quantify our observation, we collect the lifetime information of the bypassed blocks, which are chosen with the DSB bypassing algorithm [21], and report the lifetime histogram of selected benchmarks in Figure 4.2. Here, the lifetime is measured as the number of LLC misses while a cache block was live in the L1 cache (i.e., the number of LLC misses between the time when the bypassed block is allocated in the L1 cache and the time of its last touch before being evicted). From Figure 4.2, we can see that bypassed blocks quickly become dead in the
L1 cache. For example, for benchmark \textit{art}, 75\% of its bypassed cache blocks have a short lifetime of between 3 to 4 LLC misses in the L1 cache and 96.4\% of its bypassed blocks are dead after 8 LLC misses. On average of all the benchmarks in our study (see Section 4.4 for methodology), 94.3\% of the bypassed blocks are dead after 8 LLC misses. We also collected the lifetime information of the bypassed blocks in the L2 cache and it exhibits very similar trends.

![Figure 4.2: The lifetime histogram of the blocks, which are bypassed from the LLC, in the L1 data cache.](image)

The implication of the lifetime information on inclusive LLCs is that those blocks, which would have been bypassed otherwise, are essentially useless and are allocated in LLCs only for the inclusion purpose. Note that even marking those blocks as early victims to evict in LLCs may still replace more useful data, thereby not as effective as bypassing. In the next section, we leverage the short lifetime of bypassed blocks to design our low cost solution to enable cache bypassing for inclusive LLCs.

4.3. Adaptive Cache Bypassing for Inclusive LLCs

To enable cache bypassing for inclusive LLCs, we propose a bypass buffer (BB). The bypassed blocks are kept in the BB rather than replacing victims in an LLC. When a block is evicted from the BB, it invalidates the copies of the same data in upper level caches to ensure
the inclusion property. The lifetime information presented in Section 4.2 shows that the bypassed blocks become dead quickly in L1/L2 caches. Therefore, a small BB is sufficient to reap the performance benefit of bypassing while maintaining the inclusion property.

Next, we present our design to incorporate a bypassing algorithm within an inclusive cache hierarchy. We use the winning algorithm from CRC [80], Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing (DSB) [21]. A key feature of DSB is bypassing the LLC adaptively, which is shown as the highest contributing factor to the performance gains. Then, we show how the proposed BB can be leveraged to reduce the hardware cost of the DSB algorithm. Our design is based on an inclusive LLC (L3 cache) and a non-inclusive L2 and L1 caches shown in Figure 4.1d, as used in Intel Core architectures [83].

4.3.1. Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing (DSB)

In this section we briefly present the DSB algorithm and summarize the key ideas [21].

1. A Segmented LRU (SLRU) replacement algorithm [37], which was originally proposed for cache management for disk systems. Random promotion and aging policies are proposed to enhance performance.

2. An adaptive bypassing policy, which randomly bypasses cache blocks based on a probability. This probability is increased or decreased based on whether bypassing is effective or not. The effectiveness of bypassing is determined by tracking whether a bypassed block is reused before the replacement victim. To do so, each cache set is augmented with an additional tag and a competitor pointer. In the case of a bypass, the additional tag field keeps the tag of a bypassed block and the competitor pointer points to the replacement victim, which would have been evicted without bypassing. If the competitor is accessed before the bypassed one, bypassing is determined as effective. If the bypassed tag is accessed before the competitor, bypassing is determined to be ineffective. DSB algorithm invalidates a bypass block – competitor pointer pair when there is a fill at the location pointed by competitor pointer. To assess the impact of
bypassing when a no-bypassing decision is made, some newly allocated blocks are randomly selected for ‘virtual bypassing’. In this case, the additional tag keeps the tag of the replacement victim and the competitor pointer keeps the position of the newly incoming block. If the replacement victim is re-accessed earlier than the incoming block, it means that bypass is effective.

3. Set sampling, in which a few sample sets maintain auxiliary tag directory (ATD) to exercise two dueling policies and a saturating counter decides which policy is applied to the cache.

**4.3.2. Bypass-Buffer Enabled Inclusive DSB**

With a BB, we only need to make the following small changes to support inclusion.

1. If the bypassing algorithm decides to bypass a requested cache block, it is allocated in the BB instead of the LLC and forwarded to upper level caches. If the BB is full, a victim is selected and the data copies of the victim are invalidated in upper level caches.

2. L2 cache misses are serviced with both the BB and the LLC. A hit in the BB provides the data to the L2 cache and the cache block is de-allocated from the BB and filled in the LLC.

**4.3.3. Data-less Bypass Buffers**

To reduce the hardware overhead of a BB, we propose to not include payload data in BB entries. A data-less BB is feasible as the tags are sufficient to maintain the inclusion property. Since the bypassed cache blocks become dead in upper level caches very soon, a hit in the BB should be very rare. Therefore, a data-less BB does not incur performance penalties. In a case when there is a miss in the LLC and hit in BB, it is treated like a miss and the data is brought in from memory. Considering multi-processor design, the BB entries also keep coherence information along with tags similar to the LLC tag store. Assuming a MESI-like coherence protocol, a data-less BB can work without any significant modifications. For snoop requests that do not need to respond with data, the data-less BB acts exactly the same
as the LLC. In the case for a snoop request asking for data which hits in the BB with the M state, the upper cache levels are searched to find the most recent copy of the data.

### 4.3.4. Efficient Tracking Using Bypass Buffers

As discussed in Section 4.3.1, the DSB algorithm needs to track the effectiveness of bypassing and does so by adding additional tags and pointers in each cache set. This incurs relatively high hardware overhead. We propose to leverage the BB to reduce such bookkeeping cost by adding a competitor pointer in each BB entry. Since the number of BB entries is much smaller than the number of sets in the LLC, the overall storage requirement for the DSB algorithm can be significantly reduced.

As the tracking information is no longer stored in each cache set, we make the following modifications to the original DSB algorithm:

1. For each bypassed block, its competitor pointer points to the replacement victim, which would have been replaced without bypassing.

2. When a block is chosen to be virtually bypassed (i.e., it is still allocated in the LLC but selected to assess the impact of bypassing), a BB-entry is allocated for the replaced block and its competitor pointer points to the newly allocated block. Since the probability of virtual bypass is low in the DSB algorithm, we do not expect the BB to be flooded with virtual bypasses/victims.

3. When L2 cache misses are serviced with both LLC and BB, depending on whether (virtual) bypassed blocks are accessed earlier than the corresponding replacement victims, the bypassing probability is adjusted accordingly, same as the original DSB algorithm.

To summarize, we present three adaptive bypassing designs for inclusive LLCs: (1) DSB with a BB containing data (I-DSB-BB-data) (2) DSB with a data-less BB (I-DSB-BB) (3) DSB with a data-less BB, which is augmented for tracking bypass effectiveness (I-DSB-BBtracking). Since the data stored in the BB are very rarely accessed, we mainly focus on I-DSB-BB and I-DSB-BBtracking in the rest of the discussion.
The design of the bypass buffer used for I-DSB-BBtracking is shown in Figure 4.3. It is organized as a set associative structure of multiple BB-entries. In each entry, the BB-tag is the block address of the bypassed block. It is different from the tag stored in the cache because the index bits are removed from the cache tags in any cache. To track the effectiveness of bypassing, a virtual bypass bit and a competitor pointer are maintained in each BB-entry.

![Set Associative Bypass Buffer](image)

**Figure 4.3: Various fields present in a BB-entry**

### 4.3.5. Hardware Overhead of Bypass Buffer

Here, we discuss the hardware storage of bypass buffer for I-DSB-BBtracking. In most of our experiments (if otherwise not mentioned), we use a 64-entry BB which is organized as a 4-way set associative structure. Each entry contains a 54-bit (= 64-bit address – 6-bit block offset – 4-bit index) tag field, a competitor pointer and two status bits as shown in Figure 4.3. Since the tag field shares the same index bits for the LLC as the competitor (i.e., the bypassed block and the competitor are in the same cache set), the competitor pointer is reduced to a way pointer. For a 16-way LLC, a competitor pointer requires 4 bits. So, the overall hardware storage cost of the BB is 64× (54+4+2) = 3,840 bits.

In comparison, the original DSB algorithm keeps a 16-bit partial tag for bypassed block, a competing way pointer (4 bits for 16-way set associative cache) and 2 status bits. As a 2MB cache with 64-B blocks has 2048 sets, the overall cost is 22×2048 = 44K bits. Therefore, I-DSB-BBtracking incurs 91% less hardware overhead compared to DSB cache bypassing algorithm.
For a 4MB shared LLC in a 4-core system, we use 256 entry bypass buffer. The storage cost of our BB-based design is 256*(52+4+2) = 14.5K bits whereas the original DSB implementation costs 88K bits of storage.

The auxiliary tag directory and randomization hardware as proposed in DSB remain the same in I-DSB-BB and I-DSB-BBtracking and they cost 46.8K bits for a 2MB LLC (93.5K bits for a 4MB LLC) and 51 bits, respectively [21].

4.4. Experimental Methodology

To model the performance impact of our proposed approach, we use an in-house execution-driven simulator. This simulator uses the SimpleScalar [7] frontend while the timing simulator is completely revamped to model a 4-way issue superscalar processor with a 64-entry active list. The memory hierarchy contains a 32kB 4-way set associative L1 data cache with a block size of 64 bytes (1-cycle hit latency), a 32kB 2-way set associative L1 instruction cache (1-cycle hit latency) and an 8-way set associative 256kB L2 cache with a block size of 64 bytes (10-cycle hit latency). We use 16-way set associative 2MB LLC with a block size of 64 bytes (30-cycle hit latency) for our single-core systems. For multi-core systems, we increase the capacity of shared LLC to 4MB. The LLC in our baseline system is inclusive and enforces inclusion on L1 and L2 caches by sending back invalidations for LLC evictions. L1 and L2 caches are kept non-inclusive as mentioned before. The main memory latency is 200 cycles.

We include all the SPEC 2000 and SPEC 2006 benchmarks that we were able to compile and run using the SimpleScalar ISA (PISA), 16 from SPEC 2000 and 7 from SPEC 2006. We use reference input for all the benchmarks and use Simpoint [29] tool to find simulation phases. For each benchmark, we use a representative 100M Simpoint for simulations. We also include 4 additional memory intensive phases and label them as gap-2, gcc-2, mcf-2 and sphinx-2. Among the 27 benchmark Simpoints we use in this study, we only report results for 14 selected programs phases. The selection criterion is that either these phases show performance gains, measured with instructions per cycle (IPC), of more than 3% when the
LLC size is increased to 16MB from the baseline size of 2MB or they have more than 5 LLC misses per 1K instructions (MPKI).

To evaluate our proposed design in a 4-core system, we generate four multi-programmed workload categories: (a) 4H: all 4 benchmarks with high MPKI; (b) 3H1L: 3 benchmarks with high MPKI and 1 benchmark with low MPKI; (c) 2H2L: 2 benchmarks with high MPKI and 2 benchmarks with low MPKI; and (d) 1H3L: 1 benchmark with high MPKI and 3 benchmarks with low MPKI. We do not include category 4L in this study because of its low memory intensiveness. In each category, eight multi-programmed workloads are generated randomly. The performance of multi-programmed workloads is measured using the weighted speedup as proposed in [59].

4.5. Experimental Results

4.5.1. Effect of bypassing on LLC performance

We start our experimental analysis with evaluating the LLC miss rates obtained by the baseline system, DSB with a non-inclusive LLC, I-DSB-BB with an inclusive LLC and I-DSB-BBtracking with an inclusive LLC and the results are shown in in Figure 4.4. I-DSB-BB and I-DSB-BBtracking both use a 64 entry bypass buffer which is organized as a 4-way set associative structure. DSB is able to reduce LLC misses for many benchmarks. For some benchmarks such as equake, mcf, parser and sphinx, I-DSB-BB has slightly more misses than DSB. It is caused by inclusion victims, i.e. few live L1 and L2 blocks being invalidated due to back invalidations. Between I-DSB-BB and I-DSB-BBtracking, some entries in a 64-entry BB are evicted early, which affects the accuracy of tracking the bypassing effectiveness for I-DSB-BBtracking. Therefore, I-DSB-BBtracking has a slightly higher number of misses than I-DSB-BB. This difference gets smaller as we increase the number of BB entries (see Section 4.5.3 on the impact of the BB size). Also, the benchmark mcf from SPEC2000 (mcf-2k) have a very low number of LLC misses for a 2MB LLC therefore there is no impact of using bypassing for this LLC configuration. But we include this benchmark because it shows high MPKI due to thrashing behavior when the LLC capacity is 1MB. (More results in Section 4.5.6).
Next, we analyze the fraction of bypassed LLC allocations and fraction of bypass buffer hits (BB-hits) for I-DSB-BBtracking. Figure 4.5 shows the fraction of LLC allocations which are decided to be bypassed and Figure 4.6 shows the fraction of bypassed blocks which are recalled by L2 cache and experience a hit in BB. As shown in Figure 4.5, many benchmarks heavily prefer bypassing of cache blocks. For most of the benchmarks with high fraction of cache bypassing (e.g. art, gcc-2, mcf and sphinx), the bypassing is effective and we observe significant reductions in LLC misses as shown in Figure 4.4. The exception is the benchmark equake, which shows a high fraction of bypasses and yet does not achieve significant reduction in LLC miss rate, meaning that both the bypassed blocks and their competitor LRU blocks have no reuses. The benchmarks bzip2-2k, gromacs, lbm, mcf-2k, parser and vortex show low amount of bypassing and therefore their LLC miss rates are largely unaffected. The benchmark ammp has a repetitive access pattern with very long reuse distances and causes the tracking information (i.e. bypass block –competitor pointer pair) to be cancelled before it can be detected to be effective (as mentioned in III-A). Therefore, the bypassing probability stays low, and this minimal amount of bypassing leads to a small reduction in the LLC miss rate.
A key aspect of our motivation of proposing the Bypass Buffer is that the bypassed blocks are not likely to be reaccessed by upper levels. We also mentioned in the Section 4.3.3 that hits in bypass buffer should be very rare and therefore it does not incur any performance penalty if BB-entries are data-less. In Figure 4.6, we present the fraction of hits in the bypass buffer (called BB-hits) normalized to the number of cache bypasses. There are two key observations that can be made from comparing Figure 4.5 and Figure 4.6. First, for most benchmarks with high amount of bypassing, the fraction number of BB-hits is very low. Second, benchmarks such as bzip2 (SPEC2000) and vortex have relatively higher fraction of BB-hits. A BB-hit indicates an incorrect bypassing decision and results in lower probability of bypass. Therefore the fraction of bypassed blocks is relatively low for these two benchmarks.

![Figure 4.5: Fraction of bypassed LLC allocations for I-DSB-BBtracking](image)

![Figure 4.6: Fraction of bypassed blocks incurring a hit in the bypass buffer for I-DSB-BBtracking](image)
4.5.2. Performance improvement of Bypass Buffers

In this experiment, we evaluate the effectiveness of our proposed I-DSB-BB and I-DSB-BBtracking designs. We present the performance improvements, measured in the instruction per cycle (IPC) speedups, as shown in Figure 4.7. For reference, we also show the IPC improvement of the non-inclusive DSB design. From Figure 4.7, we can see that DSB achieves an 11.6% IPC improvement for non-inclusive cache hierarchies on average, using the geometric mean (Gmean), across the high MPKI benchmarks. Both I-DSB-BB and I-DSB-BBtracking enable bypassing for inclusive LLCs. I-DSB-BB achieves an average IPC speedup of 9.8% while I-DSB-BBtracking has an overall speedup of 9.4%. As discussed in Section 4.3.4, I-DSB-BBtracking uses the BB to keep usage information for both bypassed blocks and the replacement victims chosen to participate in virtual bypass. Compared to I-DSB-BB, some entries in a 64-entry BB are evicted early, which affects the accuracy of tracking the bypassing effectiveness. Therefore, I-DSB-BBtracking has slightly lower performance than I-DSB-BB. When increasing the BB size to 128 entries, the performance gains of I-DSB-BBtracking is improved to 10.0%. Considering its significant savings in hardware cost and relatively minor performance difference to I-DSB-BB, we consider I-DSB-BBtracking as our design of choice.

![Figure 4.7: Performance improvements of DSB, I-DSB-BB and I-DSB-BBtracking (w.r.t. the baseline inclusive LLC with the LRU replacement policy)](image_url)
4.5.3. Effect of the Bypass Buffer size

In this section, we analyze the performance of I-DSB-BB and I-DSB-BBtracking for different bypass buffer sizes. As mentioned in Section 4.2 the number of bypass buffer entries required should be small because of the lifetime of most bypassed cache blocks in upper cache levels is small. Therefore, we chose to experiment with a design with 64 BB-entries. Figure 4.8 shows the geometric mean of IPC speedup of benchmarks for different sizes of bypass buffer. It can be observed from the results clearly that increasing the size of bypass buffer increases the performance of I-DSB-BB gradually. On the other hand, I-DSB-BBtracking gains performance with increasing size of bypass buffer more rapidly in the beginning but it saturates after 128-entries.

To elaborate, increasing the bypass buffer size for I-DSB-BB allows the tags of bypassed blocks to be stored in BB longer. Therefore, with the increase of size of bypass buffer the performance increases. I-DSB-BBtracking has two benefits from increasing the size of BB. The first is the same as I-DSB-BB and the second is that the bypass tracking mechanism has more entries and the detection of effectiveness and ineffectiveness of bypassing is done more accurately. This is the reason why I-DSB-BBtracking recovers more performance compared to I-DSB-BB when number of BB-entries is increased.

![Figure 4.8: Performance of I-DSB-BB and I-DSB-BBtracking for different bypass buffer sizes](image-url)
4.5.4. Comparison to a high performing replacement algorithm, DRRIP

DRRIP [34] is one of the high performing cache replacement algorithms for LLCs. It provides scan-resistance and thrash-resistance via the use of bimodal insertion policies and set dueling [63]. We adopt the source code (distributed by the authors) to incorporate DRRIP in our simulator framework. We compare the performance gains of the DRRIP replacement policy with our proposed I-DSB-BBtracking design and the results are shown in Figure 4.9. In this experiment, both DRRIP and our I-DSB-BBtracking are used for an inclusive LLC and the baseline is an inclusive LLC with the LRU replacement policy. For reference, we also include a bar for each benchmark in the figure showing the IPC speedup of DRRIP on a non-inclusive LLC and it is normalized to same baseline system as other two bars (inclusive LLC with the LRU replacement policy).

From Figure 4.9, we can see that both DRRIP and our proposed I-DSB-BBtracking can support the inclusion property while achieving performance gains over the baseline LLC, an average of 9.4% and 6.7%, respectively. For benchmarks such as art, gcc-2, mcf and sphinx, I-DSB-BBtracking outperforms DRRIP significantly although DRRIP already shows good performance. The reason is that DRRIP still needs to allocate a block even if it knows the thrashing behavior. Therefore, in our 16-way set associative LLC, out of 16 ways in a cache set, one way is being thrashed while other ways enjoy hits when being reused. Bypassing eliminates such inefficiency and can fully utilize the 16 ways for data reuse. The other limitation of any replacement algorithm, as discussed in Section 4.2, causes both program phases of the benchmark sphinx to degrade performance compared to non-inclusive case. Multiple consecutive accesses to the same cache set are inserted at the LRU position in the set and evict each other in the case of this benchmark. This phenomenon does not hurt performance in a non-inclusive LLC but it degrades performance when inclusion is enforced via back invalidation. As a result, DRRIP (inclusive LLC) shows 17% and 7% lesser performance for sphinx and sphinx-2 respectively compared to DRRIP (non-inclusive LLC). On the other hand, I-DSB-BBtracking recovers all the performance in the case of sphinx-2 and ensures no slowdown in the case of sphinx. This recovery of performance is enabled by the bypass buffer which lets the bypassed cache blocks stay in the LLC for longer duration as
opposed to using DRRIP which evicts the LRU inserted cache blocks on the next miss to the cache set.

**Figure 4.9:** Performance improvements of DRRIP and I-DSB-BBtracking (w.r.t. the baseline inclusive LLC with the LRU replacement policy)

4.5.5. Performance gains of Bypass Buffers in the presence of a stream-buffer

High performance microprocessors employ hardware prefetching mechanisms to hide memory latency. For our high MPKI benchmarks, when we employ a stream buffer [36] with the following configuration: 8 four-entry stream buffers with a PC-based two-way 512-entry stride prediction table, the streaming buffer prefetcher results in a 39% IPC improvement on average. Here, it is interesting to see whether the intelligent LLC management can still benefit in the presence of the stream buffer. As shown in Figure 4.10, when we use I-DSB-BBtracking algorithm for inclusive cache hierarchy with such a stream buffer, a 9.2% IPC improvement (on average) is observed over the baseline inclusive cache hierarchy with the LRU replacement policy and the same streaming buffer.

In comparison, inclusive DRRIP provides an IPC speedup of 7.3% in such a case. Therefore, we conclude that bypassing for inclusive cache using I-DSB-BBtracking can
outperform intelligent replacement policy like DRRIP in presence of stream prefetching as well.

![Graph showing performance improvements of DRRIP and I-DSB-BBtracking in presence of a stream buffer. The baseline is an inclusive LLC with the LRU replacement and a stream buffer prefetcher.]

**Figure 4.10:** Performance improvements of DRRIP and I-DSB-BBtracking in presence of a stream buffer. The baseline is an inclusive LLC with the LRU replacement and a stream buffer prefetcher.

### 4.5.6. Sensitivity to LLC configurations

We also evaluate DSB, I-DSB-BB, I-DSB-BBtracking and DRRIP for different LLC sizes and set-associativity. Their average performance gains across the high MPKI benchmarks are shown in Figure 4.11. The baseline LLC for each set of bars is of specified size and uses LRU replacement policy. The same BB size, 64, is used for all these cache configurations. The rationale behind the same size of bypass-buffer being effective for various cache sizes is that the bypass-buffer is sized for the lifetime of bypassed cache blocks in upper cache levels and which does not change significantly with the configuration of LLC. Moreover, the number of cache sets (which increases with increasing cache size and decreasing set associativity) increases the overhead of bypassing for DSB and I-DSB-BB while I-DSB-BBtracking has a fixed overhead.

From Figure 4.11, we can see that the adaptive bypassing schemes achieve high performance for all the cache configurations that we studied. As we move towards a smaller capacity of LLC, the performance gains of bypassing (DSB) as well as replacement (here
DRRIP) are reduced. This is due to the thrashing behavior of some of the benchmarks which causes the baselines of 1MB and 2MB to be similar performing. On the other hand, DSB and DRRIP both provide thrash resistance and 2MB LLC can fit bigger portion of the working set compared to 1MB LLC and enjoys significantly more hits. It should also be noted that, with bigger LLC capacity of 4MB the difference in performance between DSB (non-inclusive LLC) and DRRIP (inclusive LLC) grows comparatively small. This is due to two reasons. Firstly, the negative effect of enforcing inclusion decreases when LLC capacity is bigger. Secondly, DSB does not evict data from LLC while DRRIP has to insert data in LLC which hurts more performance for smaller cache associativity/capacity and vice-versa. Overall, we can see that I-DSB-BBtracking consistently outperforms DRRIP across a variety of LLC configurations we studied.

![Figure 4.11: Performance improvements of DSB, I-DSB-BB and I-DSB-BBtracking for different cache configurations. (the baselines are inclusive LLCs with the corresponding configurations)](image)

4.5.7. Bypass Buffers for Shared Last Level Caches

In the next experiment, we focus on the effectiveness of the BB for the shared LLCs in multi-core systems. For a 4MB LLC shared among 4 cores, we employ a 256-entry BB and measure the performance of 32 multi-programmed workloads as described in Section 4.4.
Although the original DSB bypassing algorithm is inherently thread unaware, it outperforms thread-aware DRRIP (TA-DRRIP) [34] in our experiments. We did not observe any significant improvement from making DSB algorithm thread aware therefore we conclude that a simpler (thread unaware) design is a more cost effective approach. We also compared the performance of a shared BB and a private BB in our design. A thread private bypass buffer limits the amount of bypassed blocks a thread can store in the bypass buffer. Moreover, if there are a few threads in a multi-programmed workload that do not prefer bypassing, many BB-entries may be underutilized. In our experiments, the shared bypass buffer design performs superior to a private bypass buffer design and therefore it is the choice in our design of bypass buffer for shared LLC.

Our simulations results comparing the performance of TA-DRRIP and I-DSB-BBtracking are reported in Figure 4.12, which shows that our proposed I-DSB-BBtracking algorithm provides an average speedup of 6.6%, 11.6%, 8.4%, 4.3% for the category 4H, 3H1L, 2H2L and 1H3L respectively. The workloads in these categories have one or more programs, which have high MPKI and they compete heavily for the shared LLC. The bypassing algorithm selectively bypasses the cache blocks and therefore provides the cache capacity to data blocks with smaller reuse distances.

![Figure 4.12: Performance improvements of I-DSB-BBtracking and TA-DRRIP for a 4MB inclusive LLC for multi-programmed workloads (w.r.t. the baseline with the LRU replacement policy)](image-url)
As shown in Figure 4.12, TA-DRRIP also improves the performance with an average of 4.8%, 9.4%, 7.2% and 2.9% for categories 4H, 3H1L 2H2L and 1H3L respectively. Due to strict allocation policy, it always has to allocate a data block in cache. Moreover, multiple insertions from different threads at lower LRU stack positions in a cache set result in victimizing the blocks prematurely and a loss of performance due to inclusion. This is responsible for relatively lower average performance of TA-DRRIP compared to I-DSB-BBtracking algorithm in all the categories. Therefore we conclude that bypass buffer can make cache bypassing effective for inclusive shared LLCs as well.

4.5.8. Energy Consumption

In this section, we compare the energy consumption of I-DSB-BBtracking scheme with the baseline. We use McPAT [44] to obtain the static and dynamic power consumption results. The power model is based on the 45nm technology and 3.4 GHz frequency. Figure 4.13 shows the energy consumption for each benchmark when the 100M simulation phase is executed on the baseline system as well as when the I-DSB-BBtracking mechanism is deployed.

![Figure 4.13: Comparing energy consumption of I-DSB-BBtracking with the baseline](image)
From Figure 4.13, we can see that for most benchmarks DSB-BBtracking reduces overall energy consumption by up to 25.6% and 6.2% on average. Most of the energy savings are a result of reduced execution time which translates into significant savings in static energy consumption. For the remaining benchmarks, on which DSB-BBtracking has little performance impact, the energy consumption impact is nearly negligible.

4.5.9. Memory Bandwidth Reduction

As shown in Section 4.5.1, I-DSB-BBtracking reduces LLC misses. This in turn reduces the off-chip memory traffic and the execution time. As the reduction in LLC miss-rate is typically higher in proportion than the reduction in execution time, average memory bandwidth is reduced. As shown in Figure 4.14, we observe average memory bandwidth of 3.3 GB/s in the baseline for benchmark *art* while I-DSB-BBtracking reduces it to 1.9 GB/s. For benchmarks such as *bzip2*, *equake*, *gromacs* and *mcf*, we observe slight increase in the bandwidth. *Bzip2* and *gromacs* have slightly more misses compared to the baseline and therefore the bandwidth is increased. On the other hand, *equake* and *mcf* have reduced number of LLC misses and yet the bandwidth increases due to faster execution of the program. Overall, the aggregate memory bandwidth is reduced by 13% on average across the benchmarks.

![Figure 4.14: Comparing memory bandwidth of I-DSB-BBtracking with the baseline system](image-url)
4.5.10. Additional Benefits of Cache Bypassing Algorithms

Many proposals for high performance cache management focus on replacement algorithms. A common theme is to alter the insertion policy whereas the allocation policy is strict and each incoming block is allocated space in an LLC. The key contribution of a bypassing algorithm is to combine a placement / allocation policy with the replacement policy. Bypassing has some interesting benefits for upcoming memory technologies. For example, phase-change memory (PCM) provides high integration density while suffers from limited write endurance. In a design using PCM technology for LLCs, we can utilize cache management algorithms like DSB to reduce the number of fill operations to the LLC. This would not only increase the life of such PCM based LLC structure but also increase the overall performance (as shown in previous sections). In Section 4.5.1, Figure 4.5 shows the fraction (in percentage) of bypassed LLC allocations when the I-DSB-BBtracking algorithm is used. From this figure, we can see that for certain benchmarks, up to 80% of the LLC allocations can be bypassed while enjoying good performance gains.

4.6. Related Work

There is a plethora of research work on designing high performing cache replacement algorithms. Recently proposed replacement algorithms [9][34][51][70] for LLCs focus on providing thrash-resistance and scan-resistance. There is also a significant amount of work on cache bypassing algorithms [11][21][22][35][38][43][54][63][71]. The results of 1st JILP Cache Replacement Competition indicated that bypassing algorithm like DSB [21] can be an effective method to improve the LLC performance for a wide variety of workloads. Recent work by Li et al. [43] uses a similar mechanism to decide the effectiveness of bypassing. The key difference is they do not use probabilistic bypassing and instead maintain a signature based history per program counter to make bypassing decisions. A major disadvantage of using signature based approach is to have the program counter being sent to LLC from each core with the cache access. This can be an expensive requirement to fulfill in a real design. Therefore, DSB is our choice of bypassing algorithm in this study. However, it should be
noted that a bypass buffer can help employ any complex bypassing algorithms on inclusive LLCs.

Gaur et al. [22] proposed cache bypassing algorithms specific to exclusive cache hierarchies. The key contribution of this work is to tackle the problem of no reuse information being available for cache blocks (a cache hit de-allocates the cache block and the reuse information is lost). The work presents both insertion and bypassing algorithms designed for exclusive caches.

Jaleel et al. [33] pointed out a key performance problem in inclusive cache hierarchies due to invalidation victims. The non-inclusive cache performance is achieved in this work by making the last-level cache replacement policy aware of the temporal locality in upper levels of cache. This work can be used in conjunction with our scheme to further enhance performance for inclusive caches.

4.7. Summary

In this work, we focus on the inherent limitation of inclusive caches in utilizing cache bypassing. We propose and evaluate a novel solution, called a bypass buffer (BB), to overcome this limitation. The bypassed data blocks skip the LLC and their tags are stored in the BB. When a tag is replaced from the BB, it invalidates the upper cache levels to maintain the inclusion property. We show that for a well design bypassing algorithm a relatively small BB is sufficient to reap most of the performance gains of bypassing. Our proposed BB also enables us to significantly reduce the storage hardware cost of bypassing algorithms as it readily provides the usage information of bypassed cache blocks. Our experimental results show that our proposed design achieves high performance and outperforms a recently proposed high performing replacement algorithm, DRRIP, in both single core and 4-core systems. Our evaluation of our proposed design on different cache configurations and in presence of a stream prefetcher shows that it provides a cost-effective design for inclusive LLCs.
Chapter 5

Conclusion and Future Work

In this dissertation, a new mathematical model for locality is proposed and various properties and strengths of the model are discussed in depth. We show that our probability based quantitative measure of locality has clear mathematical meaning and relationship with other previously proposed models. We demonstrate its effectiveness in deriving insights at different cache levels and drive cache hierarchy optimizations. A GPU algorithm to speed-up the computation of locality measure for large access traces is also presented.

Thereafter, we propose online locality monitoring framework to estimate the locality measure using a simple hardware mechanism. Online locality monitors are used to drive dynamic cache reconfiguration for private last-level caches in single-core processors using Hcache algorithm, and dynamic cache partitioning for shared last-level caches in multi-core processors using SLCP algorithm. We show that our proposed Hcache reconfiguration algorithm is able to choose suitable operating configuration for private last-level cache that improves system performance as well as reduces cache energy consumption. Using our SLCP partitioning algorithm, heterogeneous cache partitions are achieved for a shared last-level cache which enables more effective cache sharing. We show that previous proposed
capacity partitioning algorithms are limited to exploring only one dimension of the cache partitioning problem and hence they have limited performance impact. Experimental results show that our proposed SLCP partitioning algorithm achieves significant performance gains compared to other related work. In conclusion, our unified approach to monitor spatial and temporal locality is able to make better cache partitioning decisions and leads to superior system throughput as well as reduced overall energy consumption.

In the last part of this dissertation, we have presented a solution to combining cache bypassing algorithms with inclusive cache hierarchies called *Bypass-buffer*. For an inclusive cache hierarchy with bypass-buffer, top-performing DSB bypassing algorithm can successfully achieve gains close to non-inclusive cache hierarchy with DSB bypassing algorithm. We also show that bypass-buffer can augment the bypassing algorithm by reducing its overall hardware cost significantly.

For future work, locality analysis can be used to study joint optimization of a multi-level cache hierarchy. We have demonstrated that the interplay of different cache optimizations at different levels of cache causes interesting trade-offs in optimizing the cache hierarchy, and hence motivates an in-depth study of such interactions. For GPGPU (General-purpose computing on graphics processing units) programs, our locality measure can be used to analyzing locality and gaining insights into GPU memory hierarchy. It is particularly interesting because interleaved access patterns from execution of multiple threads, thread-blocks and streaming multiprocessors present non-trivial access patterns for analysis. The author has presented a preliminary study in [28] with a case study on matrix multiplication.

Overall, we find that proposed locality measure can help analyze and reveal many interesting locality characteristics in cache and memory access behavior of various applications and our probabilistic locality measure is a powerful tool to understand and optimize memory hierarchy in offline as well as online fashion.
REFERENCES


[80] 1st JILP Workshop on computer architecture competitions (Cache Replacement Championship). http: //jilp.org/jwac-1/


[83] Intel’s Core i7 processors. http://techreport.com/articles.x/15818
APPENDICES
Appendix A

Locality Curves for SPEC2006 Applications based on CRC framework

Here, we present the locality curves for SPEC2006 benchmark application traces for L3 cache accesses (LLC), obtained from Cache Replacement Championship (CRC) framework. The traces are obtained by forwarding 1B instructions and then running 100M instructions (Figure A.1) or running 1B instructions (Figure A.2).
Figure A.1: Locality plots for L3 access traces for SPEC2006 benchmarks
Figure A.2: Locality plots for L3 access traces for SPEC2006 benchmarks
Appendix B

Locality Curves for SPEC2006 Applications based on DPC framework

Here, we present the sub-trace locality of L2 (LLC) demand accesses in presence of prefetcher (Dimitrov et al. [14]) for various SPEC2006 benchmarks. For six other SPEC2006 benchmarks similar sub-trace locality has been presented in Section 2.4.4 (Figure 2.13 and Figure 2.14). The traces are obtained from Data Data Prefetching Championship (DPC) framework by forwarding 1B instructions and then running 100M instructions (Figure B.1).

We also present the locality curves for selected SPEC2006 benchmark application traces for L2 cache accesses and L2 cache misses in absence of prefetching (Figure B.2 and Figure B.3). We only show the benchmarks that have at least 50,000 L2 misses (i.e. MPKI greater than 0.5).
Figure B.1: Sub-trace locality of L2 demand access traces for SPEC2006 benchmark
Figure B.2: Locality of L2 demand accesses and L2 miss traces in absence of prefetching
Figure B.3: Locality of L2 demand accesses and L2 miss traces in absence of prefetching