The Dissertation Committee for **Jiajun Wang**
certifies that this is the approved version of the following dissertation:

Reuse Aware Data Placement Schemes For Multilevel Cache Hierarchies

Committee:

Lizy Kurian John, Supervisor

Earl E. Swartzlander Jr

Andreas Gerstlauer

George Biros

Mohit Tiwari

Reuse Aware Data Placement Schemes For Multilevel Cache Hierarchies

by

Jiajun Wang

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2019

Dedicated to my family

Acknowledgments

I would like to express my special appreciation and thanks to my advisor Prof. Lizy Kurian John for her tremendous guidance of my PhD study. I would like to thank her for the endless patience and encouragement to allow me grow up my research interest, for the immense knowledge to guide me in all the time of research, and for continuous support to help me in not only writing of this dissertation and all the other research papers. I could not have imagined having a better advisor and mentor for my PhD study, and my respect goes to her.

Besides my advisor, I would like to thank the rest of my dissertation committee: Prof. Earl E. Swartzlander, Prof. Andreas Gerstlauer, Prof. George Biros and Prof. Mohit Tiwari, for their insightful comments and constructive criticism to help me widen my research from diverse disciplines and improve this dissertation. I would also like to express sincere gratitude for the opportunity to collaborate with my internship mentors at ARM memory research team, Wendy Elsasser and Prakash Ramrakhyani. Thank you for your constructive comments and unfailing feedback in improving my research topic.

My PhD time will lose so much fun without the interaction with other graduate students. I would like to thank all the member of LCA research group: Dr. Jee Ho Ryoo, Dr. Michael Lebeane, Dr. Reena Panda, Shuang

Song and Qinzhe Wu. Specifically, thanks Reena for helping me develop many research ideas as well as the countless coffee chats we had together. I am also grateful for the course projects and leisure activities with others: Dr. Zhuoran Zhao, Dr. Yazhou Zu, Dr. Wooseok Lee, Mochamad Asri and Kishore Punniyamurthy.

I would like to thank my mom and dad for making the start of my PhD journey possible. I can not be more grateful for their unwavering support of my decisions on the turning points in life. Thanks for sharing both joy and worries and always having faith in me.

Lastly, I will not reach the destination of this PhD journey without the support of my husband, Lu Zhang. Thanks for patiently proof reading every single word of my research paper and be the loyal and critical audience of my presentation. Thanks for always staying late with me till the last minute of paper submission, regardless of whether the deadline is at midnight or early morning. Thanks for giving me both encouraging words and agreeable melancholy during the most difficult time of my PhD. This doctorate is as much his as mine.

Reuse Aware Data Placement Schemes For Multilevel Cache Hierarchies

Publication No. _____

Jiajun Wang, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Lizy Kurian John

Memory subsystem with larger capacity and deeper hierarchy has been designed to achieve the maximum performance of data intensive workloads. What grows with the depth and capacity is the amount of data movement happened between different levels of caches and the associated energy consumption. Prior art [65] shows that the energy cost of moving data from memory to register is two orders higher than the cost of register-to-register double-precision floating point operations. As the cache hierarchy grows deeper, the energy cost on the large amount of data movement between cache layers has become non-negligible. Energy dissipation of future systems will be dominated by the cost of data movement. Thus, reducing data movement through exploiting data locality becomes essential to build energy-efficient architectures.

A promising technique to improve the energy efficiency of modern memory subsystem is to adaptively guide data placement into appropriate caches

with the performance benefit and energy cost of data movement in mind. An intelligent data placement scheme should only move data blocks with future re-reference into cache. As the working set size of emerging workloads exceeds cache capacity and the number of cores and IPs sharing caches keeps increasing, a data movement aware data placement scheme can maximize the performance of cache-sensitive workloads and minimize the cache energy consumption of cache-insensitive workloads.

Researchers have noticed that exclusive caches have better performance compared to inclusive caches. However, high performance improvement is always at odds with low energy consumption. The amount of data movement and energy consumption of exclusive caches is higher than inclusive ones. A few state-of-the-art CPU caching insertion/bypass policies have been proposed in literature. However these techniques are either at great expense of metadata overhead when adapting to exclusive caches, or they focus on reducing data movement at the sacrifice of performance. On the GPU side, designing efficient data placement schemes also faces great challenge. CPU caching schemes do not work for GPU memory subsystems, because the SRAM capacity per GPU thread is far smaller than the number per CPU threads. The capacity of GPU on-chip SRAMs is too small to hold large data structures in the GPU workloads. Data with frequent reuse is evicted before it is re-referencedwhich results in high GPU cache miss rate. Keeping the above shortcomings of prior work and key limitations in mind, this dissertation focuses on improving the performance and energy efficiency of modern cache subsystems of CPU and

GPU by proposing performance and energy sensitive data placement schemes.

This dissertation first presents a data placement for multilevel CPU caches to guide data placement into appropriate cache layers based on data reuse patterns. PC is utilized as the prediction heuristic based on the observation of good correlation between memory instruction and the locality of the data accessed by the instruction. Unlike prior art that includes great overhead for meta-data (e.g., PC) transmission and storage, a holistic approach to manage data placement is presented, which leverages bloom filters to record the memory instruction PC of data blocks. The proposed scheme incorporates quick detection and correction of stale/incorrect bypass decisions and an explicit mechanism for handling prefetches. This leads to energy efficiency improvement by cutting down wasteful cache block insertions and data movement.

To overcome the challenges on the GPU side, an explicitly managed data placement scheme in GPU memory hierarchy is presented in this dissertation. In order to improve data reuse of a popular HPC application and eliminate redundant memory accesses, data access sequence is rearranged by fusing multiple GPU kernel execution. Bank level fine-grained on-chip SRAM data placement and replacement is designed based on the microarchitecture of GPU memory hierarchy to maximize capacity utilization and interconnect bandwidth. The proposed scheme achieves the best performance and least energy consumption through reducing memory access latency and eliminating redundant data movement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the era beyond the end of Denard's scaling, scaling throughput has become the driven force to scale microprocessor performance due to the inability to scale frequency. Performance improves by processing more threads concurrently using increased core counts and by employing micro-architectural techniques like SMT/SIMD. The number of threads in modern microprocessor SoCs is increasing as exemplified in 48 thread Arm based solutions [18] and 36 thread recently announced Intel solutions [43]. However, improving throughput by putting more cores together is not sustainable due to power and thermal management challenges. High energy bills of large data centers and limited battery life of edge devices have prompted efforts to make modern computer systems more energy and power efficient.

From the field of exascale supercomputer systems to the market of mobile edge devices, it has been commonly highlighted that energy dissipation of future systems will be dominated by the cost of data movement. Prior art [65] shows that the energy cost of moving data from memory to register is two orders higher than the energy cost of register-to-register double-precision floating point operations. In the server computing environment, around 28% to 40% of

total processor energy consumption of scientific applications is spent on data movement [66]. Stories are similar to the domain of mobile application processors. The power consumption of solely working on fetching data from memory is on par with the mobile processors busy executing arithmetic operations under 100% utilization [107]. Prior art shows that as semiconductor process technology scales from 45nm towards 7nm and beyond, the compute energy scales down by 6X, whereas the energy associated with moving data across chip through interconnect does not scale as much [89]. The ratio between the energy cost of data movement and computation is expected to quickly grow in the future [71, 78].

Cache systems have been designed to achieve maximum performance for the workload. Still, the total cache capacity is often under-provisioned for data intensive workloads and under-utilized for cache-insensitive workloads. On energy efficient systems, maximizing performance at the cost of designing large caches is still desired for cache-sensitive workloads. Meanwhile, the energy consumption of large caches should not be wasted when running cache-insensitive workloads. As the working set size of emerging workloads keeps growing, modern processors employ multiple levels of caches to address the "Memory Wall" between high speed processors and orders of magnitude slower main memory. In order to bridge this gap, the computer architecture field has witnessed the growing depth of cache hierarchy from adding a second level cache as the back up of L1 cache, to the three-level cache model widely used in modern commercial CPU chips, to inserting an L0 level between the pro-

cessor and L1 as in line buffers [40] or filter caches [70], to another more level of system cache serving different IPs on the same SoC package, to innovative ideas of DRAM caches [115] (e.g., die-stacked DRAM). What grows with the depth and capacity of cache hierarchy is the amount of data movement that happens between different levels of caches and the associated energy consumption. It is common that caches account for more than 50% of on-chip die area and consume a significant fraction of static and dynamic power.

Scratchpads are fast on-chip RAMs mapped into the processors address space at a predefined address range. Scratchpad RAM is explicitly managed at software level, either by programmer or compiler. Compared to cache RAM, scratchpad RAM requires smaller chip area thanks to its simplicity in the control logic and hence it consumes less energy [14]. Software using scratchpad RAM has higher memory performance predictability compared to cache RAM which is highly dynamic. Because of this feature, scratchpad is popular on safety-critical embedded systems which have to meet certain real-time constraints [144]. Scratchpads are widely used in hardware accelerators as well. GPU chips from both NVIDIA and AMD provide programmers with scratchpad memory, which is called as "shared memory" in NVIDIA's term and "local data store" in AMD's term. Scratchpad memory enables efficient GPU thread communication within the same thread block, whereas thread communication through hardware managed caches usually causes frequent misses. Scratchpad is also used in other domain specific architectures to communicate intermediate results, e.g., input and output of hidden layers in neural network [63].

In order to achieve high performance from using scratchpads, it requires programmers to restructure their code and reorder data access sequences with hardware characteristics in mind.

1.1 Problem Description

1.1.1 Challenges in CPU memory subsystems

Recent years have witnessed the rise of code and data footprints from emerging applications, whereas the CPU cache capacity per thread/core has reached limits due to the power and die area constraints. From the the first generation of Intel Core i7 chip to the most recent Intel Core i9 design, the LLC capacity has been held to a maximum of 2MB per core during the past ten generations. Due to the diminishing return of performance on allocating larger caches [117], the performance benefit is negated by the increasing dynamic and static cache power. To improve the performance of data intensive workloads, prior research has looked at redistributing the available SRAM capacity across the various levels in the cache hierarchy and shown that many emerging workloads benefit from a larger L2 size [51, 72, 19]. Several recently announced microprocessor products appear to have conformed to this recommendation [148, 147]. Opting for larger L2 sizes however, implies that there would be greater overhead to maintain the inclusive property. In prior work, relaxing the inclusion requirement of LLCs has been shown to be beneficial with 3-12% improvements reported [51, 39]. This observation motivates exclusive caches in modern CPU cache hierarchy.

Performance and power consumption are always at odds while optimizing cache hierarchies. While exclusive caches will be high-performance as suggested above, due to the inherent difference of data block placement between an exclusive hierarchy and an inclusive hierarchy, the amount of data movement in exclusive hierarchy is generally larger than that of inclusive hierarchy [126]. Please note that in this dissertation, the terminology "inclusive hierarchy" is used to describe both strictly inclusive hierarchy and the non-strictly inclusive hierarchy (which is also called as non-inclusive hierarchy in prior arts). The data movement within the non-strictly inclusive hierarchy is the same as the strictly inclusive hierarchy, except that when a data eviction happens in the lower level of cache, the same clean copy (if exists) in the upper level cache is not invalidated in the non-strictly inclusive hierarchy, whereas a back-invalidation is demanded in the strictly inclusive hierarchy. Different from inclusive hierarchies, only the top level cache of the multi-level exclusive caches is filled with LLC miss data, and the remaining levels serve as victim caches [64], which get filled upon evictions from the upper cache level regardless of cacheline dirty status. When a cache block is evicted from its current cache level, it is written back into the next lower level cache, evicting another block if necessary. If a cacheline in a non-top level cache receives a hit, the cacheline is invalidated from current level before fetched into the top level to maintain uniqueness as well as to make room for cachelines evicted from its upper level cache. A ping-pong trip between L1 and L3 includes all three levels of caches, as an L1 evict is typically inserted into L2 first and then into

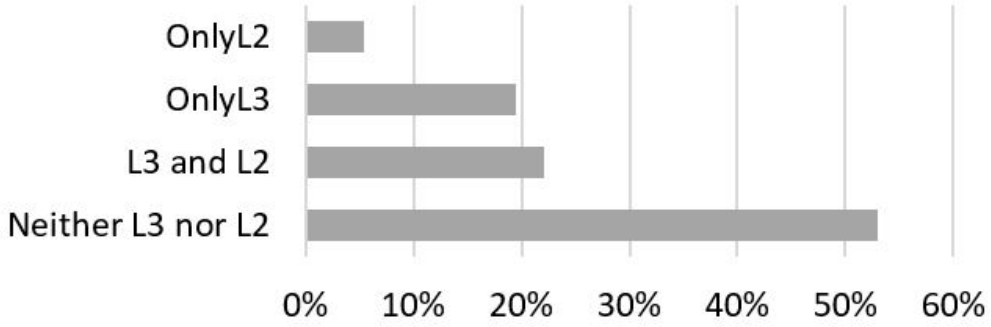


Figure 1.1: Percentage of L1 evicted cache blocks getting reused at L2 and L3 in SPEC CPU2006 (average)

L3. Large amount of on-chip bandwidth is consumed as data moves around the different levels of the cache hierarchy. There are two major causes of data movement in exclusive cache hierarchies, one is data moving from lower levels of cache to L1 due to cache hit, and the other is data moving from current level to its next level due to the eviction caused by cache replacement. Constraining data movement to L1 usually results in performance degradation because it sacrifices good L1 cacheline spatial locality and the high L1 hit rate, whereas reducing the data movement caused by eviction can be a good target for energy saving.

In order to quantify the (in)efficiency of data movement, data reuse of a dynamic cache block experiences at L2 and L3 is recorded. A dynamic data block is assumed to be a 64B (or other block size) block residing in the cache hierarchies. Dynamic cache blocks are categorized into four groups based on whether its reuse after gets evicted out of L1. The four groups are "onlyL2", "onlyL3", "L2 and L3", "neither L2 nor L3". Figure 1.1 illustrates

the distribution of reuse across all dynamic cache blocks for SPEC CPU2006 suite. From the figure it is seen that among all the dynamic cache blocks evicted out of L1, only 20% get reused in both L2 and L3. Another 5% of data blocks only get reused in L2 but no further reuse after evicted out of L2. Approximately 20% of data blocks never get reused by the time they are evicted out of L2, but are reused in L3. More than 50% of dynamic cache blocks are never reused after they are evicted out of L1. This suggests that 80% of the workload working set has optimal cache location whereas insertion into other cache levels will not bring any additional benefit. Specifically, cache blocks in the "onlyL3" category should bypass L2 when evicted out of L1 and should directly insert into L3, and cache blocks in the "Neither L3 or L2" category should write back to main memory once they are evicted from L1.

Apart from reducing non-beneficial data movement between various levels of caches, another major source of improving memory subsystem energy efficiency is to reduce main memory (DRAM) accesses. The access latency of off-chip DRAM accesses is often 5-7X of LLC access latency [43] and DRAM energy has been reported to account for more than 25% of the energy in data centers [92]. While the LLC latency can be hidden with instruction level parallelism (ILP) and out of order execution, accesses to off-chip memory incur stalls in the compute pipeline. To improve memory performance and save repeated data movement between DRAM and compute logic, various cache insertion/replacement policies have been proposed to improve cache hit rate.

1.1.2 Challenges in GPU memory subsystems

GPU has traditionally been an accelerator for graphics processing. In the past decades, due to the large amount of parallelism in the high-performance computing applications, GPU has been adopted as a general high-performance computing device (which is called as General Purpose GPU). The SRAM resource contention in GPU is extremely severe. The amount of SRAM capacity per GPU thread is less than 1KB, which is far smaller than the capacity of around 2MB SRAM per CPU thread. This is because GPU has much more threads than CPU but also much less on-chip SRAM capacity. There could be at maximum tens of thousands threads sharing GPU memory subsystem (e.g., 26624 threads in Maxwell architecture [7]), whereas the number of threads sharing the memory subsystem is limited by the number of CPU cores which can not be large. It is common that the LLC capacity per CPU core is around 2MB, and the total LLC capacity scales with the number of cores. On contrary, the capacity of GPU LLC (which is usually L2) is around 2MB large which is shared by all the GPU cores (i.e., "Streaming Multiprocessor" in NVIDIA terminology) and all the GPU threads. The design ideology behind the difference of CPU and GPU SRAM capacity per thread is that GPU, which is designed as a throughput-oriented machine, uses massive parallelism to hide latency; whereas the latency-oriented CPU uses large caches to hide latency. However, the current cache subsystem for GPU is inefficient for general purpose GPU computing. The first reason is because it results in high L2 miss rate. Although the performance loss is made up by parallelism,

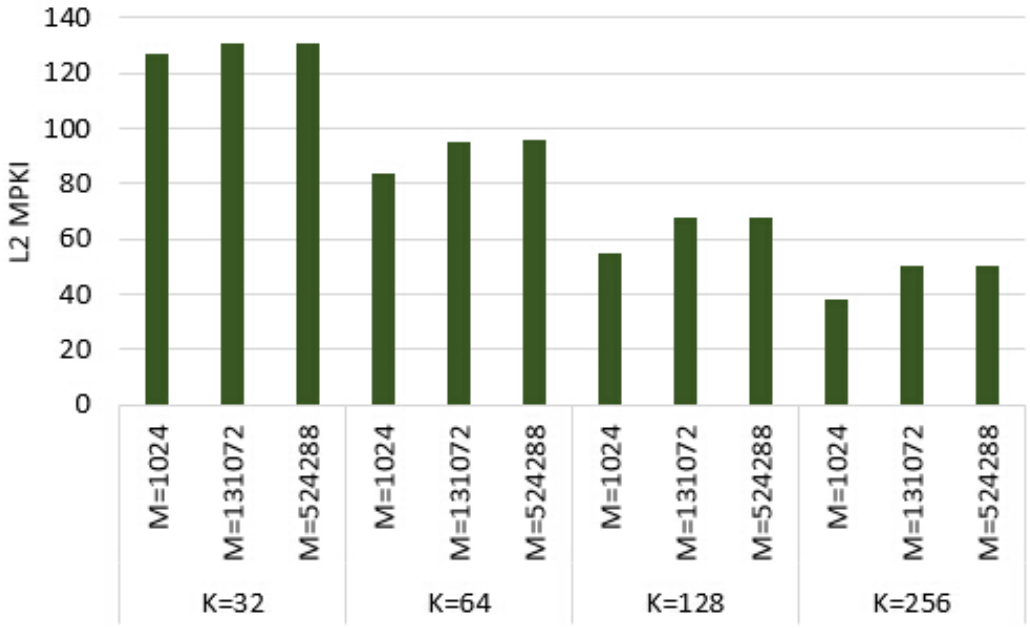


Figure 1.2: L2 MPKI of kernel summation problem, with N=1024 in all cases

the energy consumption is still huge. The second reason is because the GPU caches are too small to hold large data structures in the GPGPU workloads and the data with frequent reuse can be evicted before re-referenced due to the high contention between threads. Therefore, the data placement in the GPU cache hierarchy should be carefully handled.

Many widely used high performance computational kernels involve matrix-matrix multiplication (GEMM) and matrix-vector multiplication (GEMV) computational primitives [69]. These kernels can be decomposed into a serious of GEMM and GEMV calls with data dependencies between two calls. Basic Linear Algebra Subprograms (BLAS) provide a user-friendly interface to compute GEMM and GEMV. Different vendors provide their own highly optimized

BLAS libraries for users, including the Intel's MKL [47] and the NVIDIA's cuBLAS [102] library. These BLAS libraries are often hand-optimized with assembly code and achieve over 80% of the peak performance. Although these are good options, they still have limitations. It is observed that using black-box BLAS libraries results in performance degradation when the geometric dimension of data set size is small. The first reason is because the performance of BLAS library becomes memory bound with small data size [95]. The second reason is that state-of-the-art GPU solutions which apply cuBLAS library cannot exploit much of the data locality. Using vendor-provided libraries brings performance benefit through the highly optimized BLAS, but it also sacrifices data locality because the intermediate matrix, as the return value of GEMM call, is written back to main memory due to its huge size not fitting into caches. Figure 1.2 illustrates the number of L2 misses per kilo instructions (MPKI) when applying the cuBLAS library in a kernel summation problem. There is high L2 MPKI number in dimension $K = 32$. Since L2 is the last level cache of GPU memory hierarchy, memory performance suffers a lot from high DRAM access latency and energy is wasted on redundant DRAM accesses. The kernel summation problem typically involves large data sets, and the long memory access latency is the crucial bottleneck of program execution. Redundant and slow intermediate value accesses to main memory suggests opportunities in performance and energy optimization.

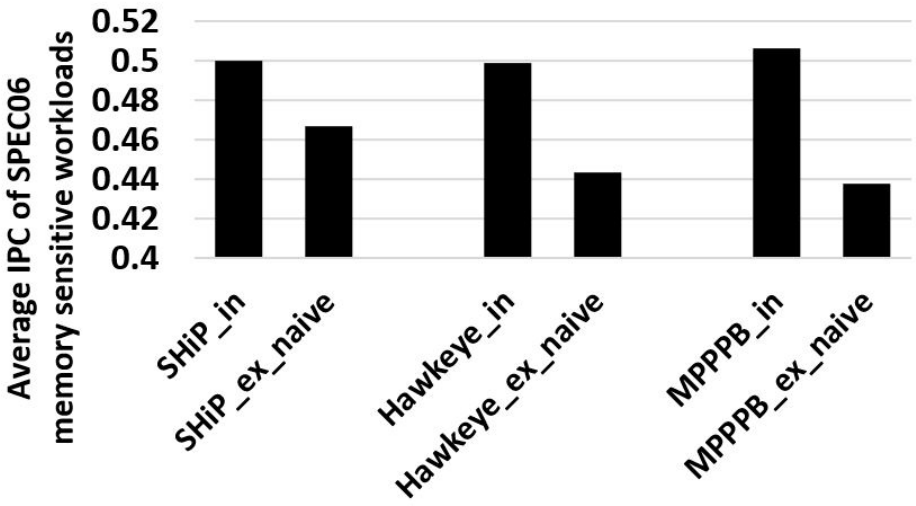


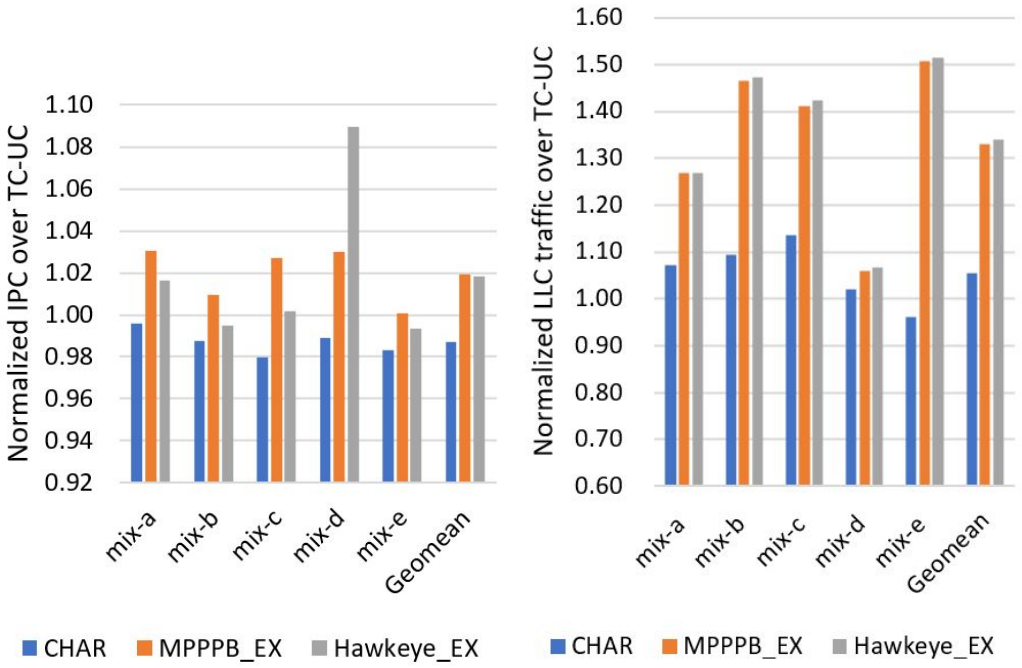
Figure 1.3: Performance of inclusive cache orientated and PC-correlated algorithms deteriorate if LLC is exclusive.

1.2 Limitations of Prior Research Work

The majority of state-of-the-art CPU caching insertion/bypass/replacement policies [52, 150, 50, 58, 33] are tuned for inclusive caches. PC-correlated algorithms are popular and yield high-performance in the field of inclusive cache design. However, a direct adaptation of PC-correlated cache bypass/replacement algorithm is ineffective. Figure 1.3 illustrates the performance comparison of three recently proposed PC-correlated schemes SHiP, Hawkeye and MPPPB [52, 150, 50, 58] between their original proposal tailored for inclusive LLC and the same design put under exclusive LLCs. The figure shows performance degradation when LLC is changed from inclusive to exclusive, which is largely due to the following two reasons. Firstly, the predictors that assume inclusive properties capture the reuse behavior of cacheline at certain cache level by observing

subsequent hits in the same level. It is non trivial to apply such techniques to an exclusive hierarchy because upon observing a hit, cachelines in exclusives caches are evicted from the lower level and promoted to the upper level. Secondly, when making an insertion/bypass/replacement decision these schemes index into their predictors using the PC of the instruction that initiated the decision. For inclusive caches this PC corresponds to the instruction that causes the cachelines to be fetched from memory, but for exclusive caches, where lines are inserted into the lower level caches upon eviction, the only available PC is the one corresponds to the instruction that causes the eviction from the upper level. This does not have a good correlation with reuse behavior of the line that is being demoted, and so the efficacy of these schemes suffers.There are no PC-correlated algorithms tailored for exclusive caches because the required PC information is not available in exclusive caches. For exclusive caches, where lines are inserted into the lower level caches upon eviction, the PC information gets lost unless it is passed along with the cacheline across all the levels in the hierarchy. This can lead to inefficient use of space and also further exacerbate the problem of data movement.

To make the performance matches with what was claimed in prior art, exclusive cache adaptations of the PC-correlated algorithms tailored for inclusive caches are devised by allowing unlimited hardware overhead to store training data (e.g., PC). The performance and data movement of three bypass and insertion algorithms for exclusive last level caches, CHAR [19], MPPPB_EX [58] and Hawkeye_EX [50], are compared using workload mixes from SPEC CPU2006



(a) Performance. (b) LLC traffic.

Figure 1.4: Normalized performance and LLC traffic of state of the art caching schemes over TC-UC.

suite. The comparison result is shown in the Figure 1.4, with IPC and LLC traffic normalized over TC-UC [39]. MPPPB_EX and Hawkeye_EX show better performance compared to CHAR, whereas CHAR generates less LLC traffic than MPPPB_EX and Hawkeye_EX. Specifically, Hawkeye_EX demonstrates a 9% performance improvement compared to CHAR in mix-d by exploiting the data locality of lbm workload, whereas CHAR shows as large as 50% less LLC traffic than Hawkeye_EX and MPPPB_EX in mix-e due to reduced data movement from L2 to LLC (i.e., L2 eviction installed in LLC) of bwaves workload. The high-performance of MPPPB_EX and Hawkeye_EX are certainly

desirable, but the low data traffic of CHAR is also advantageous.

Several designs on GPU selective data placement have been proposed [153, 55, 156]. GPU L1 cache is good candidate for bypassing because of the low L1 cache hit rate, low per-thread cache capacity and severe cache contention. Prior art uses the PC of memory instructions as indexes to predict dead blocks in L1, because the number of distinct memory instructions in a GPGPU workload is much smaller than the data size and the SIMD style instruction execution pattern makes it easy to learn data access pattern by sampling few thread groups. While these proposals save energy by preventing streaming data structure from burning cache power and hence improve performance by reducing cache pollution, they have performance limitations in the area of high performance computing applications, especially dense linear algebra problems. The first limitation is that prior art fails to exploit the temporal locality of large frequent accessed data structures. Scientific computing applications gain performance and energy efficiency by applying fast linear algebra libraries such as GEMM routine. As the majority of scientific computing applications are designed to use GEMM as much as possible, they can be decomposed into three major phase with data input output dependencies, which are pre-GEMM data preparation, GEMM computation, and post-GEMM data processing. Due to the data intensive nature of the scientific computing applications, the size of the intermediate data passing between the application phases is usually too large to be stored on chip. While bypassing schemes could save energy by directly moving data between computation units and the main memory without

storing data into caches, performance is not improved as bypassing algorithms do not help intermediate result to be reused in cache. The second limitation is that there is little performance improvement potential left for cache bypassing scheme on the GEMM computation, which is usually the most time consuming part and is calculated by calling BLAS libraries (e.g., cuBLAS library on NVIDIA GPU). BLAS library has been heavily hand optimized with assembly language by applying hierarchical blocking on all memory levels, even including register blocking. With temporal locality being maximized in GEMM, it becomes performance insensitive to the hardware cache bypassing scheme.

1.3 Overview of Proposed Research

Facing the challenge of reducing data movement and increasing memory subsystem energy efficiency, this dissertation focuses on developing techniques that can address above challenges via intelligent data placement, which yields the performance of state-of-the-art caching schemes, but with much reduced data movement, data traffic and energy consumption.

Analyzing and understanding the inherent patterns in the memory access streams of emerging applications is essential to design efficient data placement algorithms that minimize the off-chip memory traffic and improve overall memory performance. Through analyzing the temporal and spatial locality behavior of modern scale-out workloads, it is learned that data access pattern of emerging workloads exhibit a wide range of data reuse distance. At the capacity limits, memory sub-system is under-provisioned for data intensive

workloads and over-provisioned for streaming workloads. Placing data blocks via untangling data reuse patterns saves system energy by preventing streaming data structure from moving around to waste cache capacity and burn cache power, and improves workload performance by allocating saved cache capacity to keep large frequently used data structures in cache. With the presence of prefetching scheme, workloads with long reuse distance and whose working set just fitting in cache benefit from increasing cache capacity, because large caches have high tolerance of the waste capacity on useless prefetch requests and enlarges the lifetime of early-fetched prefetch requests stayed in cache.

Most of the prior state-of-the-art data placement schemes target on improving the cache performance of inclusive caches, and they use the PC of the memory instructions as heuristics. In recent years there is a trend of moving from inclusive caches to exclusive caches because of the high performance advantage of exclusive caches. Applying prior arts on exclusive caches, however, requires PC information stored in every single cache block over the entire cache hierarchy, which costs a significant amount of training data storage and additional data movement energy as the PC moves along with the data block through interconnect. Aimed at addressing the energy cost of data movement and to devise an effective predictor for an efficient and scalable multi-level CPU exclusive cache hierarchy, this dissertation proposes a FILtered Multilevel (FILM) caching policy, which achieves good performance with reduced levels of data movement. As data blocks move around different level of caches, FILM uses centralized structure to store the PC of the access that

causes DRAM fetch, rather than holding the PC along with data blocks which requires additional overhead at every level of the cache hierarchy. Specifically, bloom filter is used to overcome the challenges associated with capturing PC-based information in exclusive caches in an efficient manner. When there is free space in the bypassed cache layer, FILM overrides the initial prediction and allows cache block insertion into the cache level achieving more low latency hits. FILM also incorporates an explicit mechanism for handling prefetches, which allows it to train differently for data from demand requests versus prefetch requests. By incorporating quick detection and correction of stale/incorrect bypass decisions, FILM significantly reduces cache block installations and data movement,

However, this approach is not sufficient to unveil the potential of reusing large intermediate data structure in GPU memory subsystem, because the GPU on-chip SRAM capacity is fairly small in the massive multithreading environment and the intermediate result between two software phases is often too large to fit into GPU on-chip SRAM. Redesigning application source code with data reuse and data placement across GPU cache hierarchy in mind is the solution to this problem. In this dissertation, kernel summation is selected as an example to illustrate the performance and energy benefit of managing data placement using scratchpad in GPU. GPU utilizes fast explicitly managed scratchpad memories, which is also called as shared memory in NVIDIA GPU, to enable inter-thread communications and hide long memory access latency. The proposed data placement scheme yields both high performance

and low energy consumption by fusing all steps of kernel summation into the GEMM code structure and optimizing memory access ordering to make good use of shared memory and cache hierarchy. Specifically, the kernel summation problem is decomposed into individual tasks with few dependencies and strike a balance between finer grained parallelism and reduced data replication. Thread to data mapping is organized in an interleaved way to achieve full memory bandwidth utilization, and the orders of accessing matrix elements from scratchpad ram is managed to avoid shared memory load and store bank conflicts.

1.4 Thesis Statement

Intelligent data placement across memory hierarchy based on their locality can reduce the amount of data movement and insertions over the memory hierarchy. This saves memory subsystem energy, reduces cache misses, and significantly improves system energy efficiency.

1.5 Dissertation Contribution

This dissertation makes several contributions to improve the energy efficiency of computing systems via intelligent data placement across multiple cache layers by taking the performance and energy effect of data movement into account. The key contributions of this dissertation are summarized below.

- The first contribution of this dissertation is a detailed memory access

pattern analysis of emerging computer workloads. It is observed that data access patterns of emerging workloads exhibit a wide range of data reuse distances. It is difficult to optimize a memory hierarchy for multiple reuse distances, as different optimizations may be needed for large or small reuse distances. Large reuse distance patterns can interfere with optimizations done for small reuse patterns. This observation motivates filtering and untangling of data reuse patterns to reduce data movement and system energy. System energy is saved by preventing streaming data structures from moving around and wasting cache capacity and cache power. Workload performance is improved by allocating saved cache capacity to keep large frequently used data structures in cache.

- The second contribution of this dissertation is a FILtered Multilevel (FILM) caching policy for exclusive cache hierarchies. A locality filtering mechanism with bloom filters and predictors is presented to capture PC-based guidance in a multi-level exclusive cache hierarchy with minimal hardware overhead. The proposed solution learns about the correctness of bypass decisions and adaptively guides data placement into appropriate cache layers based on data reuse patterns. The solution also makes prefetch aware training/learning of bypass/placement decisions. The proposed scheme demonstrates significant energy efficiency improvements and reduction in on-chip data movement. FILM improves overall energy efficiency by 9%, compared to the second highest of 4% from CHAR.

- The third contribution of this dissertation is a code fusion technique for improving GPU data locality. It involves fusing of a series of GPU kernels in order to reduce data movement. With an example of kernel summation problem, this dissertation presents a fused technique that maximizes data locality using scratchpad memory. Data access sequence and placement is manually optimized based on the microarchitecture of GPU memory hierarchy to maximize GPU memory subsystem utilization via eliminating redundant data movement and main memory accesses. In order to optimize data accesses sequence, threads are mapped to matrix elements in an interleaved way, and matrix elements are reordered to avoid any load and store bank conflicts. The proposed scheme provides up to 1.8X performance speedup and 33% of total energy saving.

1.6 Dissertation Organization

This dissertation is organized as follows. Chapter 2 provides background about prior data placement schemes. Chapter 3 presents the evaluation framework used in this dissertation and explains the set of benchmarks and high-performance computing kernels that were used. Chapter 4 presents a detailed analysis of memory access pattern on modern scale-out workloads. Chapter 5 presents details of PC-correlated locality filtering approaches for exclusive cache hierarchies by exploiting data temporal locality. Chapter 6 presents an explicitly managed data placement scheme in GPU memory hierarchy for reducing data reuse distance of high-performance computing kernels

on the state-of-the-art GPU. Chapter 7 concludes this dissertation with a summary of the contributions of the dissertation and suggestions for future research opportunities.

Chapter 2

Related Work

This chapter briefly describes the metrics used to capture data locality and provides an overview of the state-of-the-art research underlying this dissertation.

2.1 Schemes for Measuring Spatial and Temporal Locality

Broadly, temporal locality and spatial locality are captured using reuse distance and stride patterns respectively in this dissertation. Good temporal locality indicates short time period between adjacency accesses to the same address (i.e., if a particular memory location is referenced, then it is highly possible that the location will be re-referenced in a near future). Good spatial locality indicates accessing neighboring memory locations within a short time period (i.e., if a particular memory location is referenced, then it is highly possible that its nearby locations will be re-referenced in a near future).

Data Reuse distance, also known as Mattson's stack distance [96] is a powerful metric to capture the temporal locality of programs. The reuse distance of a reference in a memory address trace is defined as the number of

distinct memory references between two successive references to the same location. Essentially reuse distance captures the number of intervening references between reuse of an address. If references are put into a stack, it indicates the depth at which some reused data can be located in the stack. The percentage of data references that exhibit a specific reuse distance can be computed. The distribution of such a metric for a variety of reuse distances provides an excellent picture of the potential performance of the workload with various cache sizes. Capturing the stack distance distribution essentially captures the performance of multiple cache sizes in one simulation using a single very large fully associative cache-like model. By doing so, the stack distance approach not only provides performance of caches with different sizes but indicates the total memory footprint of each workload. For example, if the workload has a combined instruction and data footprint smaller than the stack depth, the amount of valid data in the cache (assuming the cache is invalid at simulation start) represents the total memory footprint of the workload.

Strides per memory instruction (local) and memory reference stream (global) are used to characterize the spatial locality of data accesses. Local stride is defined as the difference between consecutive effective memory addresses localized per memory instruction. Local stride a good estimation of the most frequently used stride values per memory instruction and the number of memory references that it was used for. Global stride is defined as the difference between consecutive memory addresses and is used to analyze the stride-based behavior when seen across the entire global stream of memory

accesses. This approach of characterizing and portraying the stride access patterns in terms of 64-byte blocks is similar to the approach adopted by Joshi, et al. [60] for SPEC CPU2000 benchmarks. Both local and global strides are computed at the granularity of 64-byte cache blocks.

2.2 Exclusive Caches and Data Replacement Policies

There have been several studies on intelligent cacheline bypass/placement. A group of researchers have studied the energy and performance impact of cache bypass on the first level cache [25, 10, 32, 135, 53], while another group of researchers focus on the last level [9, 31, 68, 82, 152, 88]. All these techniques only target single level of cache without addressing the problem from the perspective of the entire cache hierarchy. Wu, et al. propose Signature based Hit Predictor (SHiP) [150], a sophisticated cache insertion mechanism. SHiP predicts whether the incoming cache line will receive a future hit by correlating the re-reference behavior of a cache line with a unique signature, such as memory region, program counter, or instruction sequence history based signatures. The SHiP implementation compared in this dissertation uses program counter as the signature. Jain, et al. propose Hawkeye [50], a cache replacement policy which learns from Belady's algorithm by applying it to past cache accesses to inform future cache replacement decisions. Hawkeye is consisted of an OPTgen algorithm which uses the notion of liveness intervals to reconstruct Belady's optimal solution for past long cache accesses, and a predictor which learns OPT's behavior of past PCs to inform eviction decisions for future

loads by the same PCs. Jimenez, et al. propose Multiperspective Placement, Promotion, and Bypass (MPPPB) [58], a technique that predicts the future reuse of cache blocks using seven different types of features to capture various program properties and memory behavior. MPPPB optimizes three aspects of cache management block placement, replacement, and bypass. The set of features used in MPPPB include data address, last miss, offset and program counter.Its predictor is organized as a hashed perceptron predictor indexed by a diverse set of features, and the final prediction result is an aggregation of many predictions taking into account each prediction's confidence.

LRU policies have been generally seen to be ineffective [15] for exclusive caches. Invalidating lower level cachelines on hit poses challenges on replacement policies which are designed with non-exclusive hierarchies in mind and make replacement structure update based on the number of cache hits. For example, RRIP [52] replacement policy learns re-reference behavior of a cacheline through attaching an RRPV per cacheline whose value implies whether the cacheline will be re-referenced in near future or distant future. Rereference Prediction Value (RRPV) indicates the distance of a block from its next access. An RRPV of zero implies that a cache block is predicted to be re-referenced in the near-immediate future. Hawkeye [50] adopts the idead of RRPV in its algorithm as well. However, these replacement policies can not be directly applied to exclusive caches, because the cacheline is invalidated on hit and the information of such re-reference is lost along with the invalidation. In the original RRIP proposal, RRPV is gradually decremented when a cache-

line sees a hit; or gets steadily incremented during victim selection till one of RRPV in the same set reaches the maximum. As RRPV value is lost on a cacheline hit due to invalidation, an exclusive LLC is unable to preserve cache lines that have been re-referenced. Jaleel, et al. [51] presented modifications required for RRIP to be applied to exclusive LLC (DRRIP_EX) by adding an SFL3 bit per cacheline and condensing the re-reference information into the SFL3 bit. Specifically, the SFL3 bit is set when a cacheline gets hit at L3. On LLC insertion, if the line was originally served from memory (SFL3 is zero), it is predicted as reuse in distant future; if the line was originally served from L3 (SFL3 is one), it is predicted as reuse in near future. This paper extends this idea to both exclusive L2 and exclusive LLC by adding an SFL2 bit. SFL2 and SFL3 are set when a cacheline sees a hit and serves the data request from L2/L3, and are reset when the cacheline is evicted from L2/L3 to make room for new blocks.

Gaur, et al. explore insertion and bypass algorithms for exclusive LLCs and propose a number of design choices for selective bypassing and insertion age assignment (TC-UC). LLC bypass and age assignment decisions are based on two properties of a block, trip count and use count.

Chaudhuri, et al. propose CHAR [19], cache hierarchy-aware replacement algorithms for inclusive LLCs and applies the same algorithms to implement efficient bypass techniques for exclusive LLCs in a three-level hierarchy. The CHAR algorithm learns the reuse pattern of the blocks residing in the L2 cache to generate selective replacement hints to the LLC.

Sim, et al. propose FLEXclusion [126] which dynamically switches between exclusion and non-inclusion depending on workload behavior. FLEXclusion shows significant data traffic reduction compared with exclusive caches and moderate performance improvement with non-inclusive caches. While FLEXclusion dynamically changes between the exclusive and non-inclusive to get the benefit of high performance of exclusive caches and low data traffic of non-inclusive caches, FILM's goal is to improve exclusive caches performance and reduce data traffic by learning bypass hints.

Sembrant, et al. present a Reuse Aware Placement (RAP) policy [123] to optimize data movement across the entire cache hierarchy. RAP dynamically identifies data sets and measures their reuse at each level in the hierarchy. Each cache line is associated with a data set and consults that data set's policy upon eviction or installation. RAP selects a group of cachelines (called learning blocks) to help adapt changes in application and instruction behavior by ignoring bypass decisions upon installation. RAP changes placement decision from bypass to install when the number of reuse of these learning blocks reaches a threshold. RAP experiences performance degradation as an incorrect bypass decision may have caused additional cache misses before it is corrected. Another factor leading to RAP's low performance is the absence of making special training effort on the prefetch requests. Moreover, RAP involves huge hardware overhead as it requires every cache block in the cache hierarchy to maintain a 12-bit large instruction pointer field.

2.3 Data Placement Involving Software Level Management

In addition to hardware support data placement, there are special ISA support for cache bypassing in commercial processors. X86 ISA provides bypass instructions for reads/writes with no temporal locality [99]. For example, MOVNTI (Store Doubleword Using Non-Temporal Hint) instruction performs an operand store operation using a non-temporal hint to minimize cache pollution during the write to memory. When executing the MOVNTI instruction, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. Similarly, ARM ISA provides LDNP and STNP instructions (non-temporal load and store) that perform a read or write of a pair of register values. They also give a hint to the memory system that caching is not useful for this data. There are similar ISA supports such as the ld.cg instruction in the GPU architecture as well. ld.cg specifies that a load bypasses L1 cache and is cached only in L2 cache and below.

Software programmers can either encode data placement hints in the application code to directly inform hardware about future data access pattern, or precisely control when to fetch a new data block and where to place data using scratchpad memory. Scratchpads are fast on-chip RAMs mapped into the processors address space at a predefined address range. Scratchpad RAM is explicitly managed at software level, either by programmer or compiler. Compared to cache RAM, scratchpad RAM requires less chip area thanks to its

simplicity in the control logic and hence consumes less energy [14]. Software using scratchpad RAM has higher memory performance predictability compared to cache RAM which is highly dynamic. Because of this feature, scratchpad is popular on safety-critical embedded systems which have to meet certain real-time constraints [144]. Scratchpads are widely used in hardware accelerators as well. GPU chips from both NVIDIA and AMD provide programmers with scratchpad memory, which is called shared memory in NVIDIA's term and local data store in AMD's term. Scratchpad memory in GPU enables threads within a thread block to communicate with each other efficiently, as communicating through hardware managed caches usually leads to frequent misses. Scratchpad is also used in other domain specific architectures to communicate intermediate results, e.g., between neural network layers [63]. While scratchpads provide powerful tool to help programmer achieve high memory performance, in order to make full use of scratchpads it often requires programmers to restructure their code and reorder data access sequences with hardware characteristics in mind.

Chapter 3

Methodology

To evaluate the effectiveness of proposed data moving management schemes proposed in this dissertation, a combination of techniques involving measurements on real hardware systems and simulations is used. The CPU and GPU simulators used in this dissertation include ChampSim, which is a trace-based cycle-accurate simulator derived from the 2nd Cache Replacement Championship (CRC2) simulator [28], and GPGPU-Sim [13], which is a widely popular research GPU simulator. Traces feed into the ChampSim simulator is taken using PIN [90], a program analysis tool with dynamic instrumentation. Measurements of cuda application performance is performed on real hardware systems and the performance data is get from performance counters with the nvprof [6] profiling tool provided by NVIDIA. Dynamic power and energy consumed by the various policies is estimated using McPAT [84] and GPUWattch [79]. In terms of workloads, this dissertation uses a wide variety workloads, ranging from big-data workloads like CloudSuite [103], to high-performance scientific kernels, to general-purpose benchmarks suites like SPEC CPU2006 [42]. The rest of this chapter presents an overview of each tool and also a description of the different workloads/benchmark suites used to evaluate the proposed schemes.

3.1 Simulation Infrastructure and Power Measurement

3.1.1 CPU performance and power measurement

This dissertation uses ChampSim, a cycle-level architecture simulator. It models the detailed out-of-order pipeline stages and memory system including cache-hierarchy and memory controller. ChampSim uses Intel's Pin tool to generate instructions traces with instruction pointer, register reads/writes, memory addresses and other information such as branch. Pin is a dynamic binary instrumentation tool and performs instrumentation at run time on the compiled binary files. The instruction traces are feed into the back-end simulation engine. The cache hierarchy of ChampSim can be configured with various levels and different parameters (e.g., cache capacity, associativity, access latency), and each caches can be configured with its different prefetching schemes and replacement policies. The default cache inclusivity of ChampSim is non-inclusion, but can be modified to exclusion. Similarly, the number of tenants sharing one cache can also be modified. ChampSim can be configured for both single core and multicore environment, and it gathers detailed statistics counters including IPC, branch performance (e.g., misprediction rate), cache performance (e.g., insertions, hits and misses of each access type) and memory performance (e.g., row buffer hits and misses per DRAM channel).

Except for using ChampSim to evaluate the performance of proposed schemes, the performance statistics gathered from ChampSim are fed into McPAT to generate power and area number. McPAT is an architectural integrated power, area and timing modeling framework with a design constraint of target

clock rate. It specifically targets on the area and power modeling framework for manycore processor, with a wide configurable range of core types (e.g., in-order or out-of-order, homogeneous or heterogeneous), uncore (e.g., interconnect and LLC), system I/O and memory controller components. McPAT provides XML interfaces to set up parameters of target architecture designs, and performance statistics captured by a performance simulator.

3.1.2 GPU performance and power measurement

This dissertation uses nvprof [6], an easy-to-use command line interface to access the processor performance counters on real GPU machines, to measure the performance of proposed schemes. NVIDIA GPU's nvprof is a profiling tool which enables data collection of a timeline of CUDA-related activities on both CPU and GPU. This profiling tool collects kernel execution, memory transfers, memory set and CUDA API calls and events or metrics for CUDA kernels.

Directly measuring power via hardware sensors is generally considered the most accurate way to measure power consumption. However, this approach is not available as the GPU related work in this dissertation is performed on the cloud server and neither external nor internal hardware power sensors are available. Therefore, this dissertation uses GPUWattch to evaluate the energy efficiency of proposed scheme. GPUWattch is a configurable cycle-level GPU power model built on top of a well-developed GPU performance simulator, GPGPU-Sim. GPUWattch estimates the power of GPU micro-architectural

components based upon the corresponding power model in McPAT simulators and incorporates new micro-architectural simulated components by extending McPAT.

3.2 Workload Description

3.2.1 CloudSuite

Cloud computing is gaining popularity due to its ability to provide infrastructure, platform and software services to clients on a global scale. Using cloud services, clients reduce the cost and complexity of buying and managing the underlying hardware and software layers. CloudSuite is a benchmark suite for cloud services. It includes six major benchmarks.

Data Serving - There has been a significant increase in the number and diversity of NoSQL database solutions since recent years. Compared with SQL, NoSQL database provides a more flexible storage model and stronger scalability to higher data set sizes/cluster sizes. Several NoSQL data storage solutions [1, 5, 8] are used as back-ups for large Web applications such as Google Earth and Facebook Inbox. In this workload, the 15GB Yahoo! Cloud Service Benchmark (YCSB) data set is used to evaluate the performance of the Cassandra 0.7.3 database. The server load is generated using YCSB 0.1.3 client [27], which sends requests with a 95: 5 read to write request ratio in Zipfian distribution.

MapReduce - MapReduce is the computational model that is able to handle large-scale analysis, cluster/filter large amounts of data processes, and

spread computation among a group of machines. These machines first perform a map function in which data are filtered, and then conduct a reduce function in which results from different machines are aggregated. This workload benchmarks a node of a four-node Hadoop 0.20.2 cluster. A Bayesian classification algorithm, which attempts to guess the country tag of each article in a 4.5GB set of Wikipedia pages, runs on it. One map task is started on one core with 2GB Java heap assigned.

Media Streaming - Thanks to high-bandwidth internet connectivity, recent years have witnessed an explosion in the accessibility to media streaming services such as YouTube and NetFilx, etc. Such streaming services take advantage of large computing clusters to process and transmit media files in diverse formats in a high speed. In this workload, the Darwin Streaming Server 6.0.3 is used. It serves videos of varying duration (from 1 min (1.6GB) to 10 min (>10 GB)) by using the Faban driver [2] to simulate the clients. The benchmark setup uses a low bit-rate video stream to shift stress away from network I/O.

SAT Solver - Symbolic execution is heavily used in hardware and software verification. Due to the complexity of this algorithm, it becomes tractable when the computation is partitioned into smaller sub-problems and distributed to the cloud where a large number of SAT solver processes are hosted. Since modern data center consists of heterogeneous machines, a worker-queue model with centralized load balancing is usually applied to re-balance tasks across a dynamic pool of unequal computer resources. Large scale computation is

adapted to the worker-queue model meanwhile minimizing communication overhead. Klee SAT Solver is an important component of the Cloud9 parallel symbolic execution engine [24]. It is set up as one instance per core. Input traces are generated by Cloud9 by symbolically executing the command-line printf utility from the GNU CoreUtils 6.10 using up to four 5-byte and one 10-byte symbolic command-line arguments.

Web Frontend - Web services should be fault-tolerant, widely-available and be of dynamic scalability. Such requirements necessitate web services to be hosted in the cloud. There are typically three roles within the web service architectures: a load balancer to distribute independent client requests, a web server to serve client requests, and middleware to store the state in the back-end database. We characterize a front end machine serving Olio, a Web 2.0 web-based social event calendar. Nginx 1.0.10 - with a built-in PHP 5.3.5 module and APC 3.1.8 PHP opcode cache - runs on the front-end machine. A backend dataset (12GB on-disk) is generated using the Cloudstone benchmark [128]. The Faban driver [2] is used to simulate clients as usual.

Web Search - Web search engines get information through indexing, which is a process associating terabytes of data found from on-line resources to their domain names and HTML-based fields. An index serving node (ISN) of the distributed version of Nutch 1.2/Lucene 3.0.1 is analyzed with content crawled from the public internet, which has an index size of 2GB and data segment size of 23GB. It mimics real-world setups by making sure that the search index fits in memory, eliminating page faults and minimizing disk activ-

ity. Clients are simulated using the Faban driver and are configured to achieve the maximum search request rate while ensuring that 90% of all search queries complete in less than half a second.

3.2.2 SPEC CPU2006 benchmarks

The Standard Performance Evaluation Corporation (SPEC) CPU2006 suite are widely used in both industry and academia. This suite covers various aspects of system design, including CPU, memory systems, and compiler optimizations. SPEC CPU2006 is made of benchmarks representing real life applications rather than synthetic kernels or benchmarks. CPU2006 has 29 benchmarks, in which 12 are integer benchmarks and 17 are floating point benchmarks. All these benchmarks are single-threaded written in C, C++ and Fortran. In a simulation infrastructure with multicores, the multi-programmed workloads are formed by running one individual instance of one CPU2006 benchmark on one core.

3.2.3 Kernel summation

Kernel summation is a technique used to approximate the interactions between two sets of points in a high dimensional space. The need for fast kernel summation methods first appeared in computational physics, for example, computing the 3D Laplace potential (reciprocal distance kernel) and the heat potential (Gaussian kernel). Kernel summations are also fundamental to non-parametric statistics and machine learning tasks such as density estimation,

regression, and classification [41, 44, 98, 121]. Linear inference methods such as support vector machines [130] and dimension reduction methods such as principal components analysis [98] can be efficiently generalized to non-linear methods by replacing inner products with kernel evaluations [11]. Problems in statistics and machine learning are often characterized by very high dimensional inputs.

There are numerous studies that have proposed scalable algorithms and high-performance implementations of fast kernel summation schemes such as treecodes [94][16], fast multipole methods [22][73], particle-mesh methods [120], Ewald sums [30], etc. These algorithms can scale to billions or trillions of points for problems in two or three dimensions. However, they do not scale to higher dimensions because they depend linearly or super-linearly on the dimension size.

Chapter 4

Data Locality Analysis and Micro-architectural Insights

The world is entering the era of big data and machine learning. A growing number of applications are working with very large data sets. Cloud computing is gaining popularity due to its ability to provide infrastructure, platform and software services to clients on a global scale. Using cloud services, clients reduce the cost and complexity of buying and managing the underlying hardware and software layers. Popular services like web search, data analytic and data mining typically work with big data sets that do not fit into top level caches. Popular DNN models have weights with size of tens to hundreds of MB [131]. Emerging RMS (recognition, mining, and synthesis) workloads have large working-set sizes greater than 16 MB on average [87].

Emerging applications typically work with significantly larger data sets and do not fit into the typical processor's top level (L1/L2) caches. With the emergence and growing relevance of several big-data application domains, analyzing and understanding the inherent patterns in the memory access streams of emerging applications is essential to design efficient memory hierarchies to optimize application performance. There have been several studies on charac-

terizing the micro-architectural and memory system performance (cache miss rates, TLB miss rates, etc.) behavior of big data workloads [35, 38, 69] on modern computer systems. Prior arts have concluded that big-data workloads form a distinct workload class from desktop workloads, and current computer systems are inefficient to run those workloads. For example, Ferdman, et al. observe that simple hardware prefetching schemes in the current commercial computer systems do not work for cloud workloads [35]. However, there is no analysis or understanding of why hardware prefetching schemes do not work, and what micro-architectural improvements can be made to increase memory performance efficiency. Jaleel, et al. notice that small L2 caches degrade the performance of server workloads whose working set size is a few multiples (e.g. 2-4x) larger than the L2 cache capacity [51], and hence advocate for large L2 caches and relaxing cache inclusiveness.

In this chapter[1], CloudSuite is taken as an example of modern scale-out workloads to present a detailed analysis of memory access pattern. This work focuses particularly on the behavior of memory accesses at the last-level cache and beyond. Spatial and temporal data locality of CloudSuite are computed to understand what is the reason that prevents certain prefetching schemes from improving CloudSuite performance, how would large caches help increase prefetch accuracy and coverage, and how to efficiently manage cache resources based on the data locality.

[1]Contents of this chapter was previously published at the International Symposium on Performance Analysis of Systems and Software (ISPASS) in 2017 [140]. I am the principle author of this work.

4.1 Experimental Setup

Table 4.1: System Configuration

Processor	16 cores,
L1 cache	64KB, 4-way associative, 64B cacheline, LRU
L2 cache	256KB, 8-way associative, 64B cacheline, LRU,
L3 cache	8MB, 8-way associative, 64B cacheline, LRU, 16 MSHR
Main Memory	DDR3_1600K, 4 channels, 1 rank/channel

Table 4.2: Workload characteristics

Benchmarks	Simulation length	LLC accesses	LLC misses
Data Serving	4 billion instructions	36,134,150	14,993,597
MapReduce	4 billion instructions	38,119,693	16,644,104
SAT Solver	4 billion instructions	33,271,637	23,254,707
Web Frontend	4 billion instructions	10,311,403	2,277,452
Web Search	4 billion instructions	22,613,857	3,830,574
Media Streaming	4 billion instructions	65,596,871	7,303,085
mcf	1 billion instructions	79,490,811	47,682,507
bwaves	1 billion instructions	23,176,132	22,677,841
tonto	1 billion instructions	158,098	35,737

A multicore system is configured with private L1 and L2 caches, and a shared last level cache (LLC). Detailed system configuration is listed in Table 4.1. The simulation infrastructure collects the program counter of each instruction that triggers the LLC access, as well as the corresponding physical memory access address, access type (e.g., Load/Store/Eviction), and the inter-instruction distance information of the LLC accesses. Six applications from the CloudSuite and three SPEC CPU2006 benchmarks with most LLC activity are chosen to illustrate differences between address patterns in SPEC workloads and CloudSuite. The representative phases of these workloads are captured

and are used to generate LLC access traces with our tracing infrastructure. Table 4.2 summarizes the simulation intervals as well as the number of LLC accesses and LLC misses of each workload.

4.1.1 Temporal locality profile

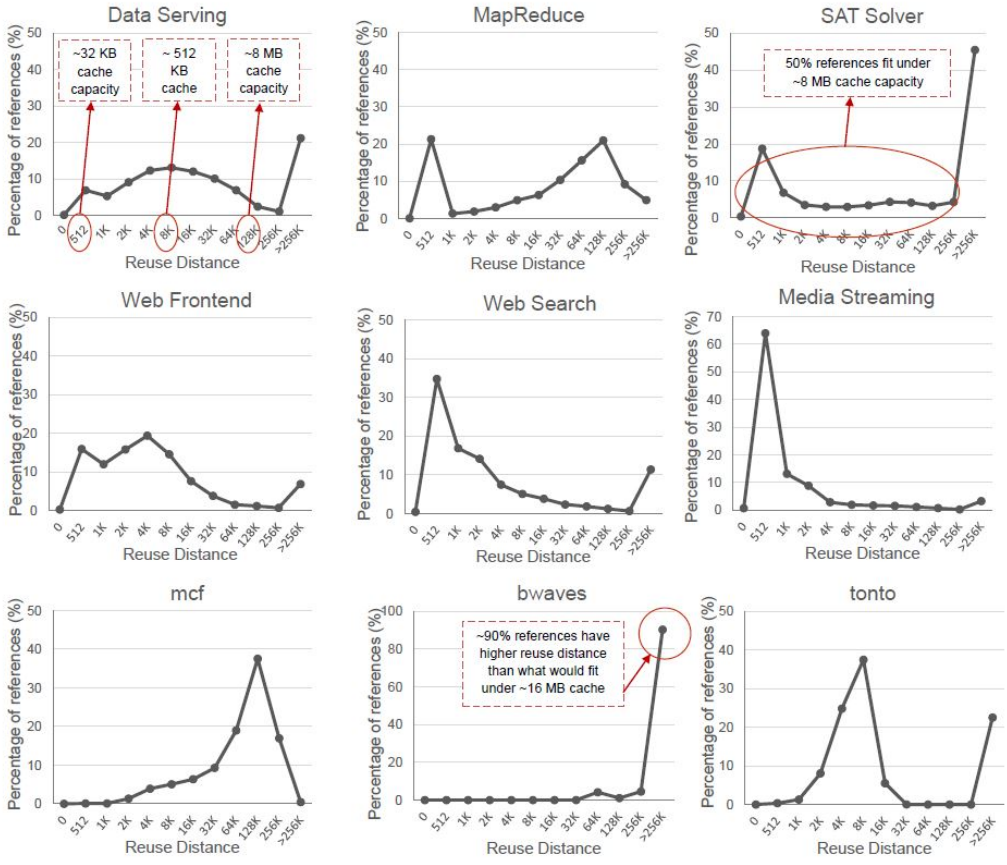


Figure 4.1: Temporal locality analysis: The figure shows the approximate reuse distance panel. Y-axis presents percentage of memory references and x-axis presents the reuse distance.

4.2 Analysis of Temporal Locality

Processor caches are designed to exploit the temporal reuse of individual memory elements to minimize the number of accesses to the off-chip DRAM memory. Temporal locality of a program dictates how the miss rate of a processors cache will change as its capacity is varied. Often the miss rate does not decrease linearly with cache capacity, but stays at a certain level and then makes a sudden jump to a lower rate when the capacity becomes large enough to hold the next important data structure. This temporal locality characterization information is represented on the reuse distance graphs for the big-data workloads and SPEC CPU2006 applications in Figure 4.1. The x-axis shows the reuse distance (from 0 to 256000, note that each x-axis point refers to a reuse distance value between itself and its previous point) and the y-axis represents the percentage of LLC accesses that have a corresponding reuse distance. Correlating the reuse distance values with an approximate cache size configuration yields that the percentage of references which have a reuse distance of say, less than 512, 8000 and 32000 will fit into a last-level cache of size 32 KB, 512 KB and 8 MB respectively.

Based on the reuse distance distribution, the nine workloads in Figure 4.1 are categorized into three types. The Web Frontend, Web Search, Media Streaming and tonto benchmarks fall into the first category, where majority of reuse distances are less than 8K. Dominant working set size of applications in the first category fits within a 2-4MB LLC. This implies that these applications have high tolerance to cache pollution from prefetching,

when used with an 8MB LLC.

The Data Serving, MapReduce and mcf benchmarks fall into the second category, where more than half of the reuse distances fall between 8K and 256K. Although more than 80% of working set fits within an 8MB LLC, cache pollution can have a detrimental impact on performance. Prefetching causes additional cache block evictions compared with no prefetching. Under non-MRU replacement policy, the chance of a cache block with longer reuse distance to become the victim block is higher than a cache block with shorter reuse distance. That is to say, when LLC capacity is just enough to hold a block with a long reuse distance in the cache until it gets reused, prefetching prevents the cache block reuse by replacing it with a prefetched line. The number of cache misses increases with the number of useless prefetch requests.

The SAT Solver and bwaves benchmarks fall into the third category, where at least half of data accesses have reuse distance larger than 256K. Applications belonging to this category have instruction and data working sets significantly large which do not fit even in a 16MB LLC. The performance impact of prefetching on SAT Solver and bwaves are completely different. Prefetching schemes can still capture address pattern of bwaves and significantly improve performance. The reuse distance in bwaves is actually infinite, indicating that there are few LLC access reuse in bwaves, and there is little negative impact on cache misses from useless prefetching.

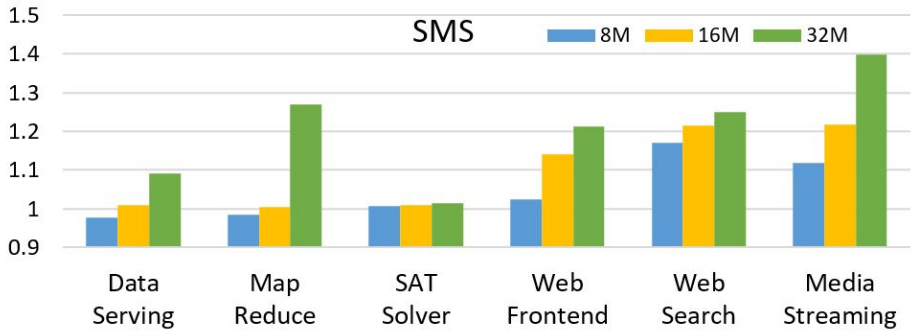


Figure 4.2: Prefetching Sensitivity of LLC Size.

4.2.1 Micro-architectural insights

Previous work shows that the LLC capacity in modern processors is over-provisioned for cloud workloads and suggests reducing capacity for power and performance [35]. However, it is observed that workloads with long reuse distance and with a working set just fitting in cache (i.e. the second category discussed in previous subsection) can benefit from increasing cache capacity. The state-of-the-art SMS prefetching algorithm is implemented as LLC prefetcher with various LLC cache capacity of 8M, 16M and 32M. The performance improvement compared to no-prefetching case is demonstrated in Figure 4.2. It can be observed from the figure that the performance of MapReduce is improved by 26% when LLC capacity doubled from 16M to 32M, whereas no performance benefit is shown when capacity increases from 8M to 16M. According to Figure 4.1, a 16MB LLC is large enough to hold 95% of MapReduce working set size, and around 60% has reuse distance longer than 8K. This indicates that a useless prefetch request has a high probability to cause

additional cache misses, because the prefetch request may evict a cacheline which becomes LRU due to the long reuse distance but is to be accessed in the future, and there is a large portion of cache blocks with long reuse distances in MapReduce. Whereas when cache capacity goes up, the chance that any cache block gets evicted decreases, and MapReduce has a higher tolerance for useless prefetch requests.

By analyzing the temporal locality of address patterns, it is discovered that there is a correlation among temporal data reuse distance of workloads, cache capacity, and system tolerance of useless prefetching. Workload performance is prone to be negatively impacted by useless prefetching when the workload has the following two characteristics. One characteristic is that the workload has a large percentage of data accesses with long temporal reuse distance. The other characteristic is that the working set of the workload is comparably large to the cache capacity. This observation can be proved on the performance increase due to cache capacity increment on a system with prefetcher making lots of early prefetch requests. Early prefetch requests are those prefetch requests which correctly predicts the future data accesses but fetches data too early into caches to be used. For example, SMS requires larger cache capacity to show the benefit of prefetching on CloudSuite workloads. As LLC capacity increases from 8M to 32M, SMS is able to bring additional 20%, 29% and 28% performance improvement for Web Frontend, MapReduce and Media Streaming respectively. Performance gain comes from an increase in the number of useful prefetch requests. Since the training and prediction pro-

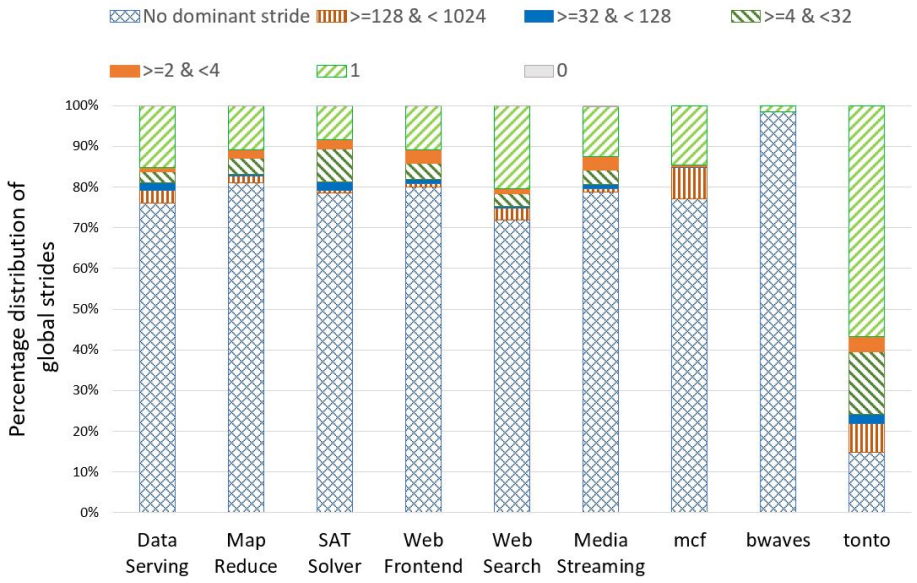
cesses are irrelevant to cache capacity, the increase in useful prefetch requests is due to fact that prefetched blocks stay longer in a larger cache as compared to a smaller cache. Therefore, the chance that a prefetched block is accessed before getting evicted goes up. For an inefficient prefetcher, e.g. a prefetchers with lower accuracy, larger cache size helps to avoid performance degradation by mitigating the performance degradation and preventing useless prefetched lines from evicting useful blocks.

Another micro-architectural insight from temporal locality analysis is the potential to optimize data placement by untangling data reuse patterns. Prior research has looked at redistributing the available SRAM capacity across the various levels in the cache hierarchy and shows that many emerging workloads benefit from a larger L2 size [51, 72, 19]. Several recently announced microprocessor products appear to have conformed to this recommendation [148, 147]. Opting for larger L2 sizes however, implies that there would be greater overhead to maintain the inclusive property. This further motivates the necessity of relaxing cache hierarchy inclusive property to exclusive property. One observation from Figure 4.1 is that the data access pattern of CloudSuite workloads exhibit a wide range of data reuse distance. For example, around 20% to 40% of data accesses in Data Serving and SAT Solver have reuse distance of either infinite distance or too long to be captured by LLC cache, whereas the rest of data accesses have short reuse distance. The data blocks residing in each level of cache hierarchies becoming mutually exclusive, and the global memory access stream consisting of data streams with different re-reference

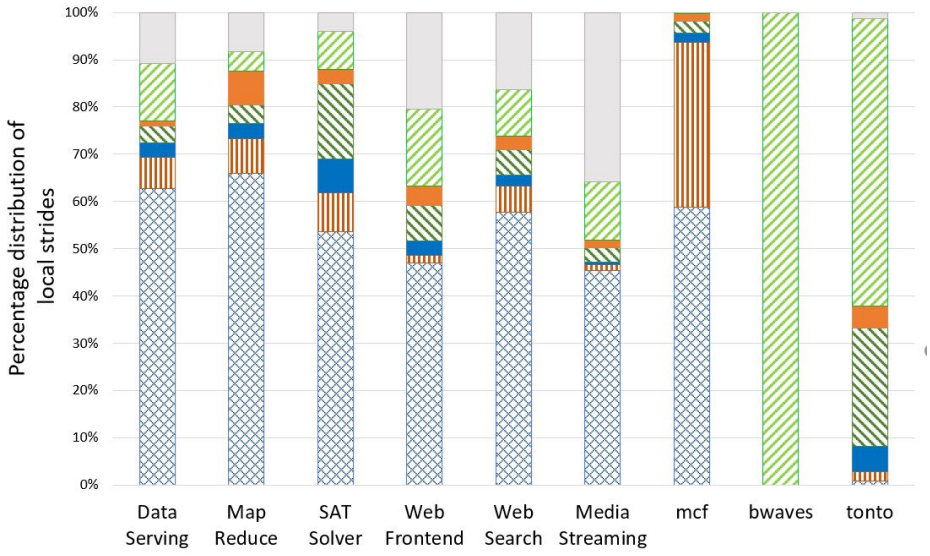
patterns, these two factors together inspire a thought of treating cache hierarchy as a sea of RAMs where data with short reuse distance is placed in smaller caches and data with long reuse distance is places in larger caches. One benefit of such reuse-aware data placement is to save energy by installing data only in levels where it will be reused. It is common that caches account for more than 50% of on-chip die area and consume a significant fraction of static and dynamic power. Regardless of whether the large cache is fully utilized by workloads or not, saving the cache energy and power cost is always beneficial. The other benefit is to improve cache utilization by preventing streaming or useless data insertion from thrashing caches. While cache capacity is at its design limit due to the power and area constraints, the capacity demand from modern workloads have been steadily increased. It becomes essential to improving cache utilization by freeing cache spaces from storing unused data.

4.3 Analysis of Spatial Locality

Spatial locality is an important characteristic of memory access patterns that is exploited heavily by prefetchers. This section presents an analysis of the spatial locality in LLC access streams of the CloudSuite workloads. A spatial locality profile is demonstrated via cumulative distribution of the most frequently used stride values and the percentage of total memory references that they make. Figure 4.3b shows the local stride distribution of the CloudSuite and SPEC workloads at a granularity of a 64-byte block, binned into categories 0, 1, 2, etc and no dominant stride categories. Figure 4.3a



(a) Global stride pattern Distribution



(b) Local stride pattern Distribution

Figure 4.3: Spatial locality analysis: Global/Local stride patterns in LLC access streams

shows the global stride distribution of the CloudSuite and SPEC workloads at a granularity of a 64-byte block, binned into categories 0, 1, 2, etc and no dominant stride categories.

It can be observed that most of the big-data workloads do not have good spatial locality at either global or per-memory instruction granularity. In terms of global locality, the most common global stride is 1, but it occurs rather infrequently (less than 15% of the time). Similarly, the local stride characterization shows that the Web Search and Data Serving workloads have a local stride of 1 for approximately 10% of the memory references, while other workloads do not possess any significant dominant local stride patterns. This behavior is expected as most cloud applications either work on data structures that have irregular memory layouts or act on random queries. On the other hand, bwaves and tonto benchmarks (both from the SPEC CPU2006 suite) have very dominant local and global stride patterns. As a result, they are very suitable candidates for prefetching solutions.

Based on the above analysis, prefetching schemes relying on detecting stride patterns of each instruction (e.g., Stride-3 [20]) are not effective for CloudSuite workloads. Such simple schemes are only able to keep track of small local strides (e.g. less than 32), which account for less than 15% of local strides in CloudSuite. Moreover, this small percentage determines the upper bound of performance improvement that stride prefetcher could achieve in an oracle situation, i.e., assuming data accesses with stride less than 32 are all missed in the baseline and prefetcher makes timely prefetching without polluting the

cache. As simple stride-based prefetching schemes fail to exploit data locality within each memory instruction, the number of prefetch requests generated are relatively small. The lower prefetch coverage brings negligible performance speedup. Hence the nature of the workloads exposes the insufficiency of those prefetching schemes.

4.4 Summary

To understand the inherent memory access behavior of scale-out workloads and provide micro-architectural insights for further memory sub-system design, this chapter includes a detailed analysis of the temporal and spatial locality of modern scale-out workloads. Data reuse distance is used to capture the temporal locality of programs. Spatial locality of data memory access patterns is characterized in terms of strides per memory instruction and memory reference stream. By analyzing the temporal locality of address patterns, the correlation among temporal data reuse distance of workloads, cache capacity, and system tolerance of useless prefetching is discovered. I also discovered that some cloud workloads exhibit both good spatial and temporal locality, i.e., they have dominant stride patterns which can be exploited by prefetching and the majority of their data accesses are with short temporal reuse distances. These workloads gain significant benefits from prefetching. Previous work shows that the LLC capacity in modern processors is over-provisioned for cloud workloads and suggests reducing capacity for power and performance [35]. However, it is observed that prefetching requires large cache capacity to show its performance

benefits, and there is a correlation between larger cache and better prefetching performance. While large cache capacity helps improve performance, it is also noticed that cache capacity per core is at its limit. Improving the performance and energy efficiency of cache system requires untangling different types of locality/reuse and identifying appropriate locations for each piece of data.

Chapter 5

Multicore CPU Data Placement Optimization

Optimizing a multilayer cache hierarchy involves a careful balance of data placement, replacement, promotion, bypassing, prefetching, etc. to capture the various properties of access streams. Often getting good performance involves aggressively orchestrating movement of the data to be available at the appropriate layers of the cache hierarchy. However it has been popularly recognized that aggressive movement of data results in high energy consumption. State-of-the-art caching policies such as Hawkeye and MPPPB yield excellent performance but incur more data movement compared to policies such as CHAR, and Flexclusion. Considering the energy cost of data movement, this dissertation proposes a **FIL**tered **M**ultilevel (FILM) caching policy, which yields good performance with reduced levels of data movement.

5.1 Proposed Scheme

This section introduces FILM which adapts PC-correlated locality filtering approaches for exclusive cache hierarchies. FILM predicts the reuse of cache blocks at each cache level, and guides evicted cache blocks to insert into the right level rather than trickle down through the various layers in the cache

hierarchy.

PC is selected as the training heuristic because good correlation between memory instruction and the locality of the data accessed by the instruction is observed. Figure 5.1 shows that the majority of active instructions which make intensive data requests in the SPEC CPU2006 workloads have stable data locality behavior at L2 and L3 of exclusive caches. A memory instruction is defined to have stable data locality at a specific cache level if more than 90% of data blocks accessed by the instruction are within the same range of reuse distance (e.g., always hit or always miss in the cache). This observation suggests that if historical data accesses made by an instruction do not benefit from caching at a given level, then future accesses from the same instruction can bypass that cache. Although PC-correlated algorithms have been proposed in prior work [50, 150, 58], they focus on inclusive cache hierarchies, where the memory instruction information is available during the data block insertion. A direct adaptation of prior art requires storing memory instruction PC along with cachelines, which introduces a significant amount of storage overhead. FILM overcomes this challenge of building the one-to-one relationship between a data block and its instruction using bloom filters as shown in Section 5.1.1.1.

Figure 5.2 illustrates how FILM is integrated into the cache hierarchy and how it closely interacts with all levels. As data blocks move around different level of caches, FILM uses centralized structure to store the PC of the access that causes DRAM fetch, rather than holding the PC along with data

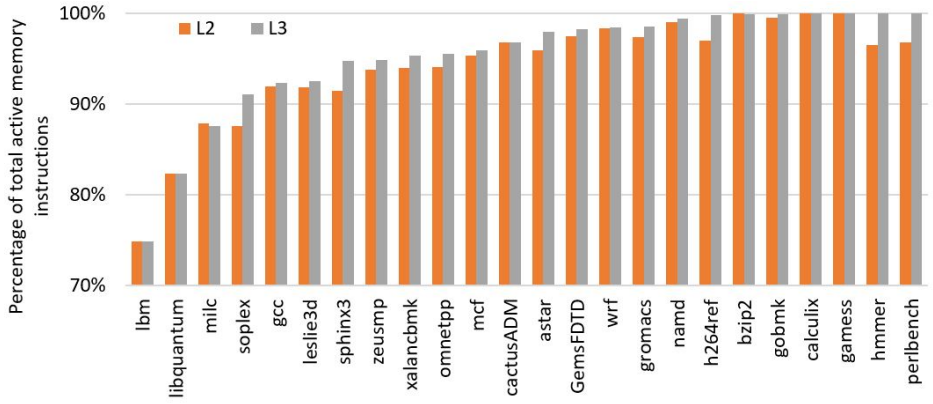


Figure 5.1: Percentage of memory instructions with stable data locality

blocks which requires additional overhead at every level of the cache hierarchy. Although FILM is a single centralized component, it is not on the critical path. A data block queries FILM about L2 and LLC bypass hints on LLC misses (activities ①) and stores hints along with the cacheline at the cost of 2-bit overhead. The access latency of FILM is orders-of-magnitude lower than the long DRAM access latency and thus can be overlapped with DRAM access latency. Training process of FILM , which does not require instant feedback, is also off the critical path. FILM's training process is triggered by the three types of cache activities shown in the Figure 5.2, ① data block installs from main memory to the top level cache due to LLC misses, ② data block hits at lower level caches, and ③ data evictions from lower level caches. By leveraging cacheline address and cacheline reuse behavior, FILM trains its prediction model to get the optimal cache insertion decision.

As shown in Figure 5.2, FILM is composed of two hardware structures, a *Prediction* table and a *Training Result* table. The Prediction Learning ta-

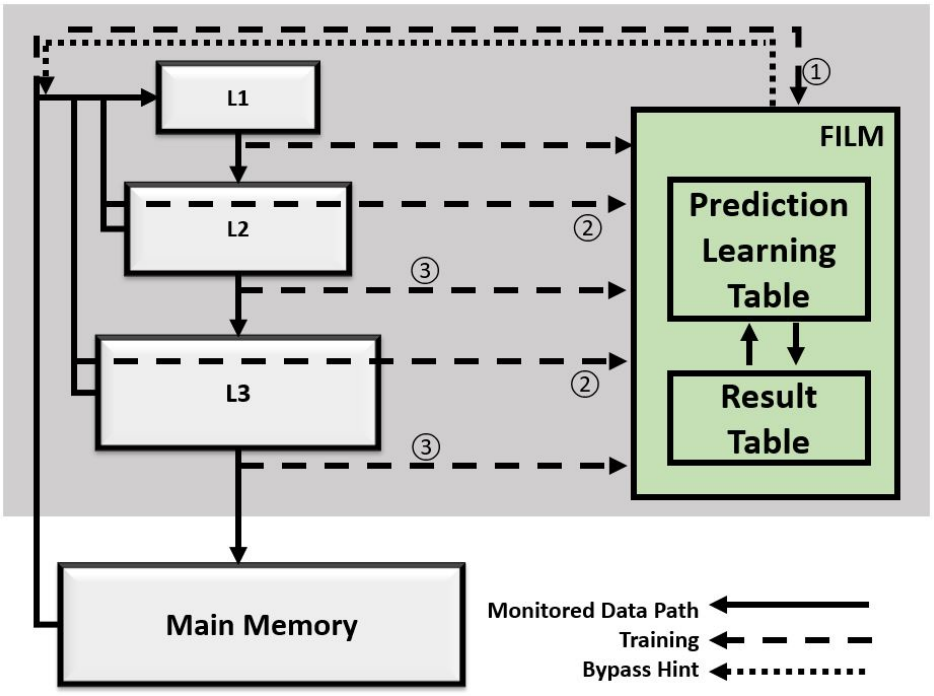


Figure 5.2: Overview of the proposed FILM system

ble learns data locality of an individual memory instruction through its data accesses history and makes bypass decisions for future data. The Result table provides bypass hints for a data block based on its memory instruction. It also records the latest training result of the instruction evicted out of the Prediction Learning table, and provides an initial value for the Prediction Learning table when an instruction is reallocated back to the table. FILM can be applied together with other cache replacement policies as FILM only provides bypass/insertion hints. The SRRIP [51] optimized for exclusive caches is used as replacement policy.

5.1.1 Handling demand requests

FILM leverages the observation that data blocks touched by the same memory instruction tend to have similar caching behavior. Thus, by learning the caching behavior of a memory instruction through access history, the caching behavior of future data blocks from the same instruction can be predicted. Building this one-to-one relationship between a data block and its instruction is not easy for exclusive cache hierarchies. One may suggest that only keeping the instruction information along with the sampled set (training set) can be a solution. However, the overhead of storing PC in sampled set is still dramatic. It requires the instruction pointer to be maintained at every cache level, and the overhead increases as the depth of the cache hierarchy grows deeper in future. For example, to sample 256 out of 8192 sets in an 8MB LLC, it requires every single core sharing the LLC to dedicate additional storage at its private L1 and L2 sets whose cacheline addresses map to the 256 LLC sampled sets. This accounts for all the L1 blocks in a 256-set 4-way 64KB L1 cache and a quarter of the L2 blocks in a 1024-set 4-way 256KB L2 cache. Secondly, it is not sufficient to store PC only in the sampled set. Because PC is required not only during the training procedure, but also during the inference (applying training result) where PC is used as an index to the Result table to get replacement/bypass hints for the cachelines belonged to non-sampled sets. Therefore, how to efficiently store the memory instruction information of data blocks becomes a crucial problem in designing PC-correlated schemes for exclusive caches.

5.1.1.1 Applying bloom filter

To address this challenge, FILM applies a bloom filter [17], a space-efficient probabilistic data structure which can rapidly determine whether a data element belongs to a data set or not. This work uses the most basic design of a bloom filter, which is in the form of a bit vector. To add an element to the bloom filter, the element is hashed a few times, and the bits in the bit vector at the index of those hashes are set to 1. To test for membership, an element is hashed with the same hash functions. The element is not in the set if any value at index bits is not set, otherwise it could be in the set.

Due to constrained training storage budget, the Prediction Learning table keeps track of a limited number (e.g.,16) of memory instructions, and every tracked instruction is assigned to a separate bloom filter. When reaching the Prediction Learning table entry limit, the instruction with the least frequent memory behavior would be evicted to make room for new instruction. The bit vector of bloom filter has a fixed size (e.g., 4098 bits). Since there is no way to delete an element from bloom filter, the chance of bloom filter reporting false positive membership increases as the number of inserted elements grows. Therefore, bloom filter is reset periodically (e.g., every 256 insertions) in order to maintain a low false positive rate. the bloom filter is named as the *local memory footprint container*, as what it records is essentially the memory footprint of an instruction.

On LLC misses, the data address is shifted by the size of a cacheline to form a cacheline address, and the cacheline address is inserted into the

bloom filter. On training events triggered by demand request hit, the PC of the memory instruction is used to index Prediction Learning table. On training events triggered by unused data eviction, FILM retrieves the PC of the memory instruction which causes the LLC miss by looking for the cacheline address among all the local memory footprint containers in a time multiplexed fashion. The searching process can be pipelined and is not on the critical path. If a single membership is reported for a cacheline address, then FILM constructs the one to one mapping between the data and the instruction, and starts the training process. Alternatively, training activities are not performed for the following two situations: one is when no residency is detected, which is possible because FILM tracks a limited number of memory instructions and local memory footprint containers get reset periodically; and the other situation is one when the address is found in more than one bloom filter due to false positive membership reports. In the latter case, FILM decides not to train to avoid training noise.

5.1.1.2 Prediction Learning table

FILM selects data blocks mapped to few LLC sampled sets as its training set. Once FILM is able to retrieve the memory instruction information of a training data block, it trains bypass heuristics for this memory instruction at all cache levels except for L1.

FILM uses a multiported table based training structure, where each entry corresponds to one memory instruction. As illustrated in Figure 5.3, each

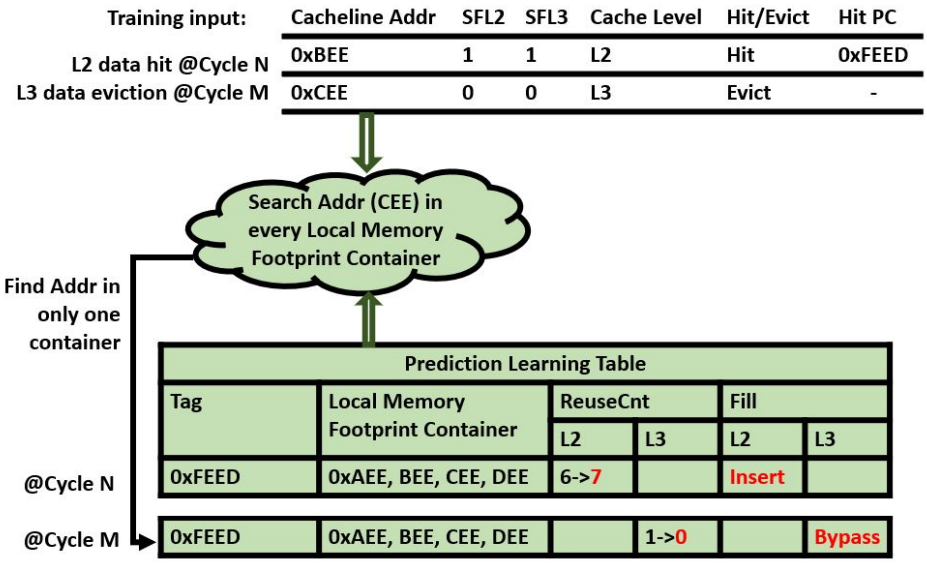


Figure 5.3: Training of FILM on demand-fetched blocks. One Prediction Learning table entry update at two different cycles.

table entry contains a *Tag* field which is used to index the Result table, a *Local Memory Footprint Container*, *ReuseCnt* fields for L2 and L3 respectively, and *Fill* fields to record current L2 and L3 insertion decisions. FILM 's Prediction Learning table updates are triggered by two activities, data reuse (hits) or block eviction with no reuse. FILM leverages SFL2 (Serviced From L2) and SFL3 (Serviced From L3) bits to indicate whether a data block has been reused during its stay at L2 and L3. Jaleel, et al. [51] presented how these bits are required for RRIP if it has to be applied to exclusive caches.

To train the prediction model, FILM requires information including PC if it is cache hits, cacheline address, reuse behavior (i.e., updated SFL2 and SFL3) and the level from which the data block gets hit or evicted. For every

instruction in the Prediction Learning table, FILM counts the number of data reuses at L2 and L3 in the table entry fields *L2ReuseCnt* and *L3ReuseCnt* respectively. The ReuseCnt field is initialized with the maximum value (e.g., 7). L2ReuseCnt and L3ReuseCnt are fixed width saturating counters, which get incremented when FILM observes the SFL2/SFL3 bit of the training input block is one, or get decremented when the SFL2/SFL3 bit is zero. The SFL2/SFL3 bit gets reset when a block is inserted into the next level of cache (i.e., L3 and memory). *L2Fill* and *L3Fill* fields record the latest training result, a bypass/insert hint. They are initialized according to the value in the Result table. An L2ReuseCnt/L3ReuseCnt value reaching the maximum value triggers the L2Fill/L3Fill field to change to "Insert", whereas value decrement to zero triggers changes to "Bypass". These operations are illustrated in Figure 5.3, where I show one example of an L2 data hit at cycle N and another example of an L3 unused data eviction at cycle M.

5.1.1.3 Result table

Upon LLC miss, data blocks consult FILM about whether to bypass L2 or LLC in future. FILM relies on the Result table to handle cases where data blocks cannot receive bypass hints from the Prediction Learning table. The Result table is a direct-mapped structure indexed by the hashed instruction pointer. Each table entry has two bits, representing L2 and L3 bypass hints separately, and their initial value are set to be "Insert". When there is an LLC demand miss, the PC of the demand request is used to index the Result

table and read the L2 and L3 fill decision. Once the data block is installed directly into L1, the decision is kept with the data block along with other metadata. The 2 bit overhead per cacheline is acceptable, and it helps guide data insertion as a complement to the Prediction Learning table. Another important function of the Result table is to provide initial Fill value for a newly allocated Prediction Learning table entry. When a Prediction Learning table entry is evicted, the trained bypass hints of the instruction are stored into its corresponding Result table entry, and such that next time when this instruction gets reallocated to the Prediction Learning table, it has warmed-up bypass hints.

5.1.1.4 Learning from bypass decisions using empty blocks

The optimal bypass hints dynamically change along with the program execution. The reason is because cache bypass of one group of data blocks changes the reuse distance of other groups. A previous bypass decision may not work in the future as the reuse distance profile changes dynamically. One example is the case when cache accesses exhibit a thrashing access pattern, e.g., a memory instruction repetitively reading K data blocks which happen to map to the same set of an N-way associative ($N < K$) cache. The optimal solution is to keep N data blocks in the cache and bypass the rest ($K-N$). An algorithm without error detection will predict that none of future data blocks from the same instruction should be inserted into cache. Whereas an optimal algorithm should allow at least N data blocks to be inserted to guarantee data

reuse at the best effort.

Wrong bypass hints are difficult to detect because the data block following the bypass hint is discarded, leaving no chance to prove its locality from cache hits. Thus, FILM is designed with a "utilize empty blocks" rule to provide opportunities to detect stale bypass decisions. The rule works as follows. Take L2 cache as an example, a data block is inserted into L2 due to available free space even if FILM suggests bypass L2. The bypass hint is stored along with this block. On a subsequent hit to the same block at L2, FILM trains its model and considers the L2 "Bypass" hint as stale after seeing that a blocked marked as bypass gets reused. In addition to increasing the L2ReuseCnt counter, FILM would immediately flip the L2 fill hint from "Bypass" to "Insert" based on the single error. Prior art either does not have error detection scheme and always perform data bypassing based on prediction, or inserts blocks if there is free space cache without any further activities on error detection.

Although the "utilize empty blocks" rule would cause useless data block insertion (which is why FILM suggests bypass), it does not cause additional performance degradation due to two reasons. One is that it does not pollute caches as it uses free cache space without causing any eviction. The other reason is that high performance cache replacement policy protects cache blocks with frequent reuse and selects cache blocks with less or no reuse as victim, such that wasted insertions from the "utilize empty blocks" rule are evicted to make room for new blocks.

5.1.2 Handling prefetch requests

FILM's training on prefetched blocks is performed at the granularity of a prefetcher. For example, for a system with L1 and L2 prefetchers, FILM sets up two entries in the Prefetch Prediction Learning table, with each entry representing one prefetcher. The local memory footprint container field is not required, because each cacheline can pinpoint which prefetcher initially fetched the block by storing prefetch level information (*PfLevel*) in the tag store. Prefetch level helps to distinguish prefetched blocks from regular blocks (whose PfLevel is zero). When a prefetched block serves a demand request, it is promoted from a prefetched block to a regular block and the PfLevel is reset to zero. The future training process on this block is handled the same as a regular block. Note that a prefetched block serves prefetch requests from upper levels does not change the PfLevel value of the block. Basically, the prefetch level of a cache block provides three hints to FILM. Firstly, it indicates whether the training target is a regular memory instruction or certain data prefetcher. Secondly, if the target is a data prefetcher, PfLevel points to the prefetcher for which FILM is to be trained. Thirdly, PfLevel being non-zero indicates that a prefetched block has not served a demand request yet.

Training on prefetch requests occurs when a prefetched block is hit by either prefetch requests or demand requests, and when a prefetched block is evicted. When a prefetched block sees a demand request hit, FILM performs three operations. Firstly, the L2ReuseCnt or L3ReuseCnt of the corresponding prefetcher is incremented based on whether the hit occurs at L2

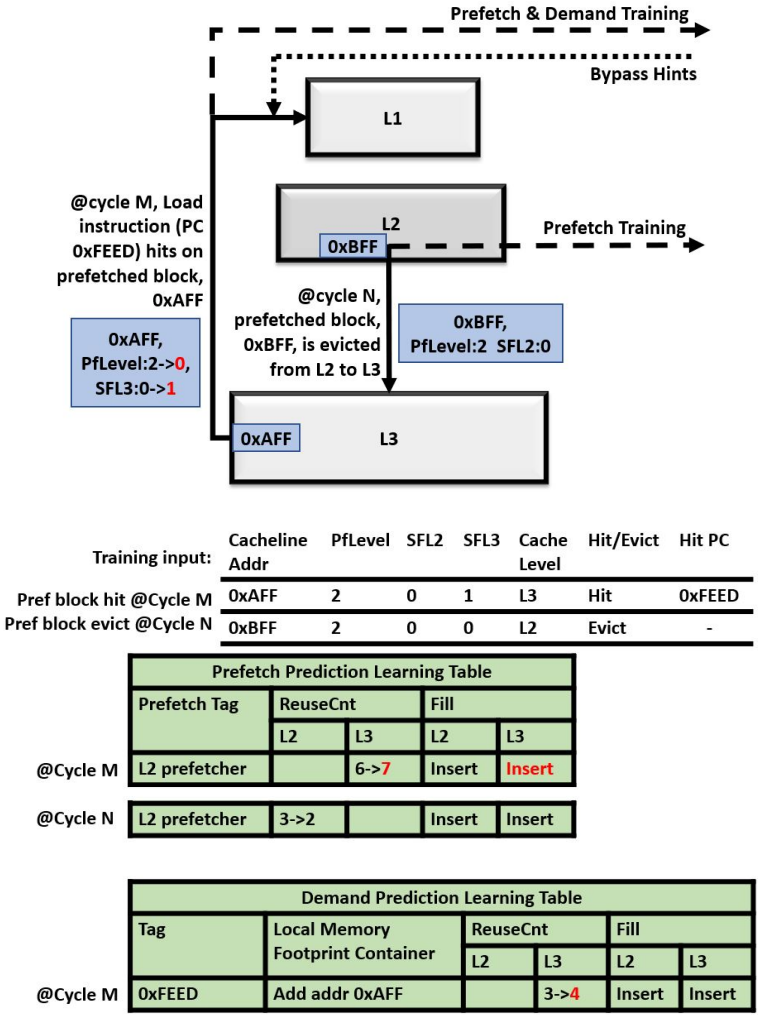


Figure 5.4: Training on prefetched blocks. Showing two different scenarios at two different cycle.

or L3. Secondly, the cacheline address of the hit block is added to the local memory footprint container of the demand request, making preparation for future training on this instruction. PfLevel of this cache block is reset. Thirdly, the demand request PC is used to consult the Prediction Learning

table and the Result table to get suggested L2 and L3 bypass hints, which is sent back to the data. When a prefetched block serves a prefetch request (hit), the L2ReuseCnt/L3ReuseCnt of the corresponding prefetcher is incremented. When a regular block serves a prefetch request (hit), the regular block is demoted to a prefetched block and has its PfLevel set, but there is no Prediction Learning table updates. When a useless prefetch block is evicted, the L2ReuseCnt or L3ReuseCnt of the corresponding prefetcher is decremented. Making L2/L3 bypass decisions and correcting stale bypass decisions are the same as handling demand requests. Figure 5.4 illustrates the above operation using two different examples. In the first example at cycle M, a data block prefetched by L2 services a demand request at L3. In the second example at cycle N, a prefetched block initiated by L2 prefetcher gets evicted out of L2 without usage.

5.2 Evaluation

Simulations are performed on a cycle-accurate simulator, which is an extended version of the 2nd Cache Replacement Champion (CRC2) simulator [28]. Table 5.1 shows detailed simulator parameters. The memory subsystem consists of a three level cache hierarchy and a detailed DRAM model. Evaluations are conducted on multicore systems with prefetching enabled. Hardware prefetching has been used for mitigating the high latency between processor and memory [141, 139]. Both traditional data prefetcher designs such as next-line prefetchers, as well as one of the state-of-the-art prefetching

Table 5.1: Simulation parameters

Four cores	out-of-order cores, 4.5GHz, 6-wide pipeline, 72-entry load queue, 56 entry store queue maximum 2 loads and 1 stores be issued every cycle
Branch Predictor	bimodal branch prediction, 16384 entries, 20 cycle mispredict latency
Private L1	64KB, 8-way associative, 8 MSHR entries RRIP replacement policy, nextline prefetcher, 4 cycle latency
Private L2	512KB, 8-way associative, 16 MSHR entries RRIP replacement policy, VLDP prefetcher, additional 8 cycle latency
Shared LLC	8MB, 16-way associative, 32 MSHR entries RRIP replacement policy additional 20 cycle latency
DRAM	4GB off-chip memory. 1 channel. 1600 MT/s Read queue length 48 per channel Write queue length 48 per channel tRP = 11 cycle, tRCD = 11 cycle, tCAS = 11 cycle

schemes, VLDP [124] are applied in this simulation. Simpoint traces of SPEC CPU2006 workloads provided by the 2nd Cache Replacement Contest [28] are used in the simulation. In multi-core experiments, cores that finish executing early would restart execution from beginning in order to continue adding pressure to shared cache and memory. In the multi-core experiments, each core runs 250M instructions with a warm up length of 10M instructions. McPAT [84] is used to estimate the dynamic power and energy consumed by the various policies. The system energy reported in this paper includes core energy, fabric energy, shared LLC energy and DRAM energy.

The FILM design is compared to seven cache replacement and bypass algorithms. **TC-UC** [39] and **DRRIP_EX** [51] learn global caching priorities for exclusive caches. Both policies use a three-bit re-reference counter per cacheline. For TC-UC, it is implemented with bypass and aging policies, which corresponds to the "Bypass+TC_UC_AGE_x8" policy in their paper [39]. Both **FLEXclusion** [126] and **CHAR** [19] focus on reducing on-chip multi-level cache bandwidth via relaxing cache inclusion policy. For CHAR the address space correlation scheme is used and the "CHAR-C4" policy is implemented which is tailored for an exclusive cache model and does not de-allocate a block from LLC on hit. For FLEXclusion, it operates in both Aggressive and Bypass mode. While above four schemes are address space correlated, code space correlated schemes, namely **SHiP++_EX** [136], **Hawkeye_EX** [50], and **MPPPB_EX** [58] are also included. These schemes are adopted for exclusive caches by storing the instruction pointer information along with data block and tailored their RRIP-based replacement policies for an exclusive cache model based on Jaleel's prior work [51]. The implementations are based on the code submitted by the respective authors to the CRC2. Specifically, the implementation of SHiP++_EX is based on SHiP++ [136], which further improves the performance of SHiP policy.

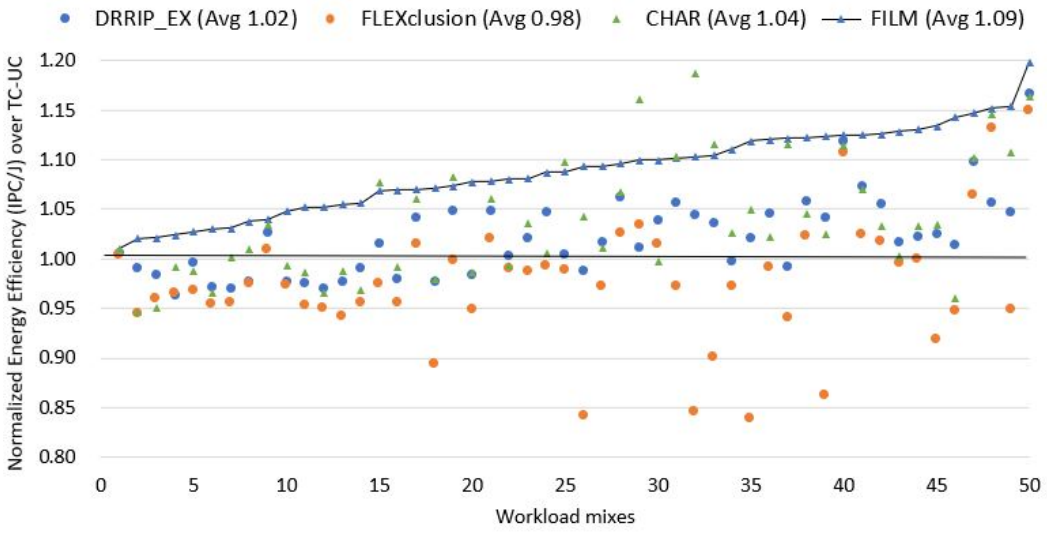
5.2.1 Evaluation results

FILM is evaluated on a series of 4-core multi-programmed workloads, with a wide variation in the data reuse characteristics and cache capacity sen-

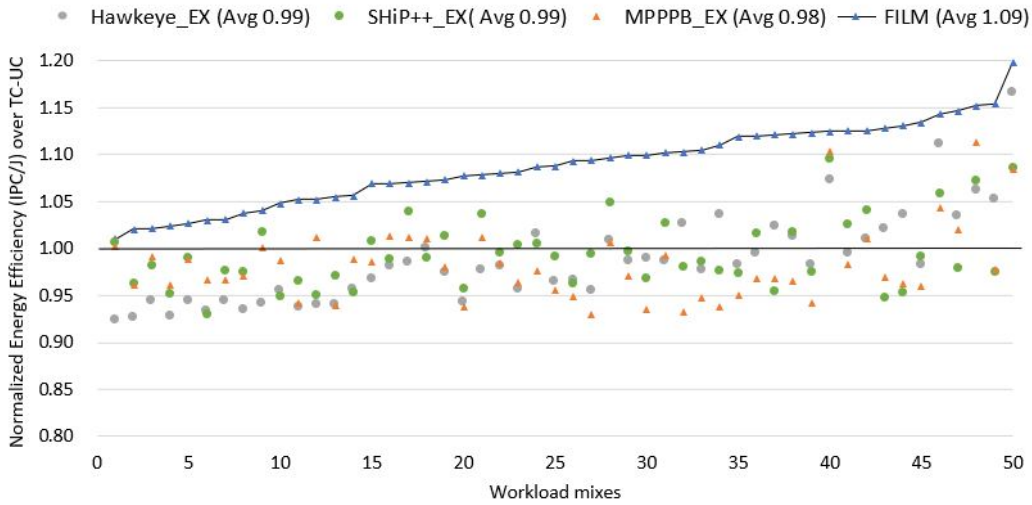
sitivity of the co-running programs. The workload mixes are made of both streaming-oriented workloads and reuse-oriented workloads. The following sections evaluate FILM and other policies from the perspective of energy efficiency, data movement and performance. An early finished workload continues executing and stressing shared resources(e.g., LLC and main memory) until the slowest one completes, however, the energy and performance of a workload is computed based on the data of first 250 million instructions. Throughput (i.e., total IPC) over entire system energy is used as the metric to demonstrate energy efficiency. LLC accesses and DRAM accesses are used as metrics to evaluate the amount of data movement controlled by the evaluated policies. LLC accesses (LLC traffic) consists of all kinds of LLC accesses, including load/store access, prefetch requests and L2 evictions. DRAM accesses (DRAM traffic) consists of all the LLC misses. IPC speedup is used to summarize the performance impact of a policy on multicore workloads.

5.2.1.1 Energy efficiency

Figure 5.5 compares the energy efficiency of CHAR, FLEXclusion, DRRIP_EX, SHiP++_EX, Hawkeye_EX and MPPPB_EX, with the number normalized to the baseline TC-UC. The mixes are arranged in the increasing order of FILM's normalized energy efficiency. Details of the workload mixes is listed in Table 5.2. The comparisons are demonstrated in two panels based on whether the policies were originally proposed to handle exclusive caches or not. The policies in the first class (CHAR, FLEXclusion, DRRIP_EX) as well as



(a)



(b)

Figure 5.5: Energy efficiency(IPC/J) of FILM and other schemes. Results normalized to TC-UC. The higher the better.

the baseline(TC-UC) are address-correlated. In contrast, all the policies in the second class (SHiP++_EX, Hawkeye_EX and MPPPB_EX) are PC-correlated

Table 5.2: Workload mixes in Figure 5.5

1.gobmk,gromacs,perlbench,wrf	26.mcf,leslie3d,gcc,milc
2.libquantum,gromacs,omnetpp,wrf	27.GemsFDTD,soplex,bzip2,wrf
3.libquantum,gobmk,gromacs,zeusmp	28.soplex,astar,gobmk,omnetpp
4.GemsFDTD,bwaves,perlbench,wrf	29.mcf,milc,wrf,xalancbmk
5.astar,gobmk,gromacs,leslie3d	30.soplex,libquantum,mcf,perlbench
6.GemsFDTD,libquantum,omnetpp,xalancbmk	31.mcf,omnetpp,perlbench,xalancbmk
7.bwaves,gcc,gobmk,xalancbmk	32.mcf,astar,bwaves,milc
8.bwaves,gobmk,gromacs,leslie3d	33.mcf,bzip2,gromacs,milc
9.gobmk,omnetpp,wrf,xalancbmk	34.libquantum,sphinx3,bwaves,bzip2
10.GemsFDTD,bwaves,gcc,xalancbmk	35.mcf,soplex,leslie3d,milc
11.GemsFDTD,bwaves,bzip2,wrf	36.soplex,mcf,bzip2,wrf
12.GemsFDTD,libquantum,leslie3d,xalancbmk	37.GemsFDTD,sphinx3,leslie3d,wrf
13.mcf,omnetpp,leslie3d,wrf	38.mcf,soplex,astar,omnetpp
14.GemsFDTD,leslie3d,omnetpp,perlbench	39.GemsFDTD,mcf,soplex,milc
15.soplex,astar,leslie3d,wrf	40.sphinx3,bzip2,gobmk,perlbench
16.libquantum,gcc,milc,xalancbmk	41.soplex,mcf,gromacs,xalancbmk
17.soplex,astar,gobmk,wrf	42.soplex,gcc,omnetpp,xalancbmk
18.libquantum,bwaves,gcc,milc	43.libquantum,sphinx3,bwaves,wrf
19.mcf,bzip2,gobmk,perlbench	44.libquantum,sphinx3,bwaves,gobmk
20.GemsFDTD,mcf,leslie3d,perlbench	45.GemsFDTD,soplex,milc,wrf
21.soplex,bzip2,perlbench,wrf	46.GemsFDTD,lbm,sphinx3,gobmk
22.soplex,libquantum,astar,leslie3d	47.sphinx3,libquantum,astar,wrf
23.soplex,gobmk,leslie3d,wrf	48.sphinx3,gobmk,wrf,xalancbmk
24.libquantum,soplex,astar,omnetpp	49.sphinx3,gcc,bwaves,wrf
25.soplex,bwaves,leslie3d,wrf	50.sphinx3,cactusADM,gobmk,omnetpp

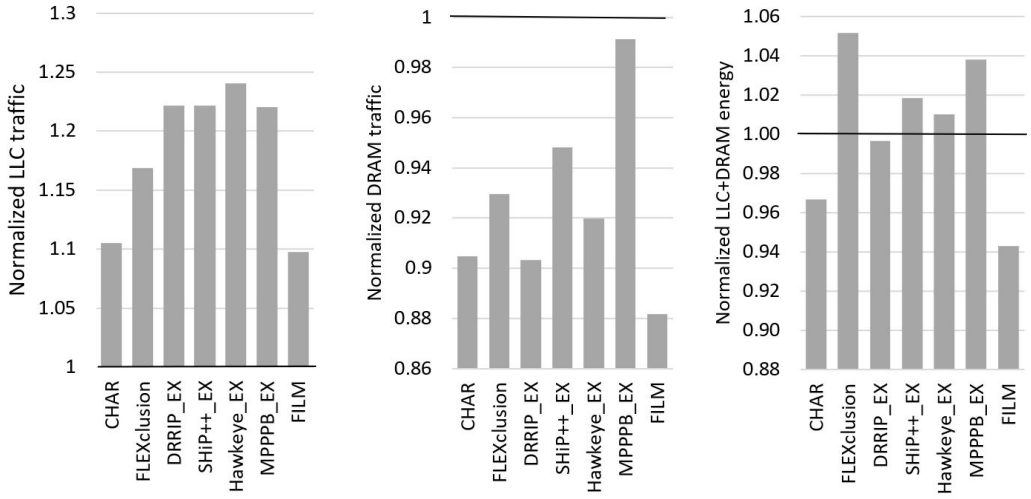
and require maintaining the program counter of the load/store instructions along with the cacheline. Additionally, CHAR and FLEXclusion, from the first group, and the baseline TC-UC policy have L2 eviction bypassing LLC mode, whereas DRRIP_EX and policies in the second group do not enable data bypassing.

One observation is that FILM constantly achieves higher energy efficiency than the baseline whereas the profile of other policies fluctuates dramatically. Compared to the baseline, the energy efficiency of FILM varies from

1% to a gain of 20%, whereas FLEXclusion shows the largest swing between a loss of 8% to a gain of 16%. my second observation is that FILM has the highest average energy efficiency improvement of 9%, beating the second largest value of 4% from CHAR by 5%. In other words, given the same amount of energy supply FILM is able to achieve 9% higher performance than the baseline, compared to the range of -2% to 4% performance improvement from other policies.

Compared to CHAR, which has the second highest average energy efficiency among all the evaluated schemes, there are only two workload mixes (i.e., mix29 and mix32) where FILM is noticeably less energy efficient than CHAR. The performance of CHAR and FILM are similar in both of the two workload mixes. The energy efficiency benefit comes from CHAR's ability to save more LLC traffic and hence higher energy saving. One major reason why CHAR has less energy consumption is that CHAR does not maintain strict cache exclusiveness, whereas FILM follows the rule of exclusive caches. Workload mixes 29 and 32 consist of some workloads which are cache capacity insensitive and other workloads whose data get repeated hits on LLC. According to CHAR's policy, a data block which has seen an LLC hit in the history will not be de-allocated from LLC on all the future hits, which violates the data exclusivity. As CHAR allows multiple copies of same data block staying in the cache hierarchy, CHAR introduces less redundant LLC data insertion of a same data block, which saves cache energy at the expense of cache capacity. As the performance of co-running programs in mix29 and mix32 is cache

capacity insensitive, the energy saving benefits of CHAR's keeping data valid on hit becomes the dominant contributor to the high energy efficiency. Except for workload mix29 and mix32, FILM has either equivalent or higher energy efficiency than CHAR, with an average of 5% better than CHAR. Specifically, FILM achieves a highest of 20% higher energy efficiency than CHAR on workload mix-46, which consists of both cache-capacity sensitive workload such as sphinx3 and streaming workload such as lbm. In this workload mix-46, FILM achieves both higher performance and lower energy compared to CHAR, because FILM enables more LLC data bypasses which on one hand saves LLC energy and capacity and on the other hand improves the performance of the cache-capacity sensitive workload. CHAR fails to explore as much bypass opportunity as FILM explores in the lbm workload due to CHAR's address-correlated learning scheme. As an address-correlated scheme, CHAR divides data into several categories based on data access type (e.g., prefetch fill, demand fill, L1 writeback fill) and cache hit status, and performs learning within each individual category. One drawback of CHAR is that data accesses of different reuse behavior can be put into the same category, impairing CHAR's learning scheme. On the contrary, FILM performs learning based on individual memory instructions. As illustrated in Figure 5.1, data accessed by the same memory instruction have similar reuse behavior. Thus, FILM has less training noise than CHAR, and is able to make more bypass decisions which contributes to FILM's higher energy efficiency.



(a) LLC traffic (b) DRAM traffic (c) LLC+DRAM energy

Figure 5.6: The traffic and energy of shared memory resource (LLC and DRAM) of FILM and other schemes. Results normalized to TC-UC. The lower the better.

5.2.1.2 Data movement

Data movement is a major factor contributing to the energy consumption difference among all the policies as it affects both LLC energy and DRAM energy. In order to understand why one policy consumes more/less energy than another, we summarize the average normalized LLC and DRAM traffic as well as the total energy of the two shared memory resources of all the workload mixes in Figure 5.6a. From the figure we could observe that FILM consumes the least amount of shared memory energy compared to other schemes because it generates the smallest average number of LLC and DRAM traffic. It is also noted that all the evaluated policies introduce more LLC traffic compared to the baseline policy with less DRAM accesses. The reason is because the

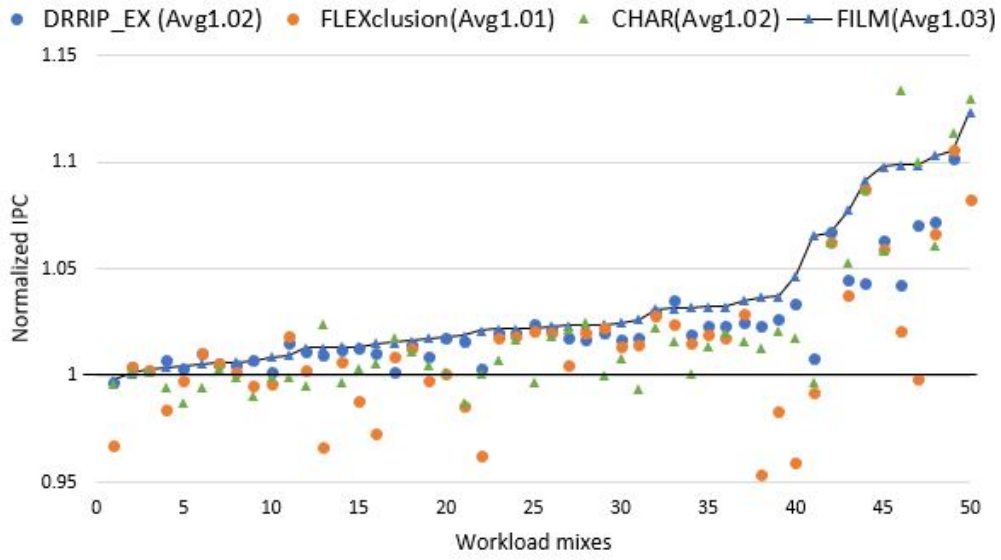
baseline policy performs LLC bypass more aggressively compared to the other schemes. Aggressive LLC bypassing helps reduce LLC energy, but results in wasting more DRAM energy due to the increasing LLC misses.

Comparing to the policies with no LLC bypassing (DRRIP_EX, Hawkeye_EX, SHiP++_EX, and MPPPB_EX), FILM saves LLC traffic by selectively installing L2 evictions into LLC. The average normalized LLC traffic for no-bypassing policies are around 1.22X, which is 0.12X more than the 1.1X of FILM. The LLC bypass rate of FILM varies between 1% to 46% (with median value of 20%), which account for up to 0.3X less LLC traffic compared to no-bypassing policies.

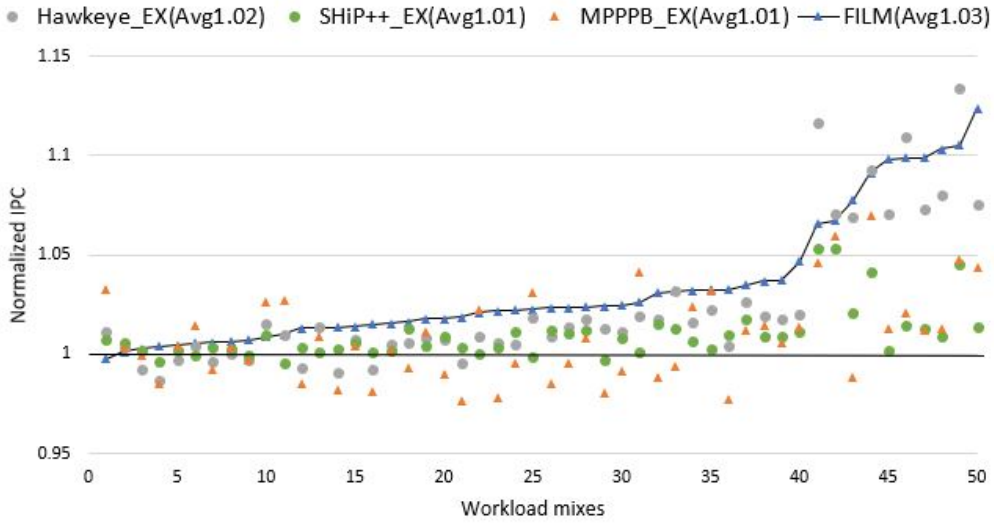
Compared with data-bypassing policies, FILM has similar average LLC traffic compared to CHAR and 0.7X less than FLEXclusion, and FILM has the lowest DRAM traffic(0.88X) compared to CHAR(0.9X) and FLEXclusion(0.93X).

5.2.1.3 Performance

Figure 5.7 summarizes the performance speedup of various algorithms normalized to the baseline TC-UC. The mixes are arranged in the increasing order of FILM's normalized throughput. Details of the workload mixes is listed in Table 5.3. The average performance of FILM is generally better than FLEXclusion and SHiP++, and looks on par with other schemes. Specifically, FILM outperforms DRRIP_EX on workload mixes with lbm and sphinx3. FILM beats DRRIP_EX in lbm and sphinx3 in terms of single core



(a)



(b)

Figure 5.7: IPC of FILM and other schemes. Results normalized to TC-UC. The higher the better.

Table 5.3: Workload mixes in Figure 5.7

1.libquantum,bwaves,gcc,milc	26.soplex,astar,gobmk,omnetpp
2.bwaves,gcc,gobmk,xalancbmk	27.soplex,astar,leslie3d,wrf
3.gobmk,gromacs,perlbench,wrf	28.soplex,bwaves,leslie3d,wrf
4.mcf,omnetpp,perlbench,xalancbmk	29.soplex,libquantum,mcf,perlbench
5.libquantum,gobmk,gromacs,zeusmp	30.soplex,gobmk,leslie3d,wrf
6.GemsFDTD,bwaves,perlbench,wrf	31.GemsFDTD,libquantum,leslie3d,xalancbmk
7.gobmk,omnetpp,wrf,xalancbmk	32.soplex,gcc,omnetpp,xalancbmk
8.bwaves,gobmk,gromacs,leslie3d	33.libquantum,soplex,astar,omnetpp
9.libquantum,gromacs,omnetpp,wrf	34.GemsFDTD,mcf,leslie3d,perlbench
10.libquantum,gcc,milc,xalancbmk	35.GemsFDTD,leslie3d,omnetpp,perlbench
11.GemsFDTD,bwaves,gcc,xalancbmk	36.GemsFDTD,soplex,bzip2,wrf
12.mcf,bzip2,gobmk,perlbench	37.soplex,libquantum,astar,leslie3d
13.mcf,astar,bwaves,milc	38.mcf,soplex,leslie3d,milc
14.soplex,mcf,gromacs,xalancbmk	39.GemsFDTD,soplex,milc,wrf
15.astar,gobmk,gromacs,leslie3d	40.GemsFDTD,mcf,soplex,milc
16.mcf,bzip2,gromacs,milc	41.GemsFDTD,lbm,sphinx3,gobmk
17.mcf,milc,wrf,xalancbmk	42.sphinx3,bzip2,gobmk,perlbench
18.soplex,astar,gobmk,wrf	43.libquantum,sphinx3,bwaves,bzip2
19.mcf,omnetpp,leslie3d,wrf	44.sphinx3,gobmk,wrf,xalancbmk
20.GemsFDTD,bwaves,bzip2,wrf	45.libquantum,sphinx3,bwaves,wrf
21.soplex,mcf,bzip2,wrf	46.GemsFDTD,sphinx3,leslie3d,wrf
22.mcf,leslie3d,gcc,milc	47.sphinx3,gcc,bwaves,wrf
23.mcf,soplex,astar,omnetpp	48.libquantum,sphinx3,bwaves,gobmk
24.soplex,bzip2,perlbench,wrf	49.sphinx3,cactusADM,gobmk,omnetpp
25.GemsFDTD,libquantum,omnetpp,xalancbmk	50.sphinx3,libquantum,astar,wrf

performance by more than 10%. FILM shows its performance advantage on cache capacity sensitive workloads by increasing the effective cache capacity via reduced insertions, minimizing shared cache capacity contention in the multicore scenario.

Among all the four PC-correlated policies, SHiP++_EX shares the most common thoughts with FILM. The largest difference is that FILM relies on bypassing dead blocks to avoid triggering the replacement policy and protecting critical data, whereas SHiP++_EX (as well as Hawkeye_EX and MPPPB_EX) always inserts dead blocks with the highest eviction priority. Take the mix50

in the Figure 5.7b as an example, FILM outperforms SHiP++_EX, Hawkeye_EX and MPPPB_EX in this workload mix, which consists of one cache capacity sensitive workload (sphinx3), one streaming workload (libquantum), and two workloads with small working set size (astar and wrf). FILM distills the streaming pattern and minimizes the LLC data replacement due to this workload, thus increasing the LLC hit rate of the data with reuse, which is retained in the cache. Another difference is that FILM detects any stale bypass decisions and updates its prediction model, whereas SHiP++_EX does not do any error detection or correction.

There is no obvious performance winner among CHAR, Hawkeye_EX, MPPPB_EX and FILM. Hawkeye_EX outperforms FILM on a few workload mixes, and its multicore throughput speedup comes from its performance improvement on one workload, lbm. However, even for the workload mixes with 8% performance difference between FILM and Hawkeye_EX, FILM achieves the same energy efficiency as Hawkeye_EX as FILM generates 25% less LLC traffic.

Co-running application could experience IPC reduction due to the negative interference between the multiple tenants contending for shared cache resource. Compared to the baseline, FILM is able to restrict the performance degradation of single application within 2%, whereas other policies lead to single application performance degradation of 4% to 10%.

Figure 5.8 shows the fairness of all the evaluated policies relative to the baseline. The minimum workload speedup is used as a conservative fair-

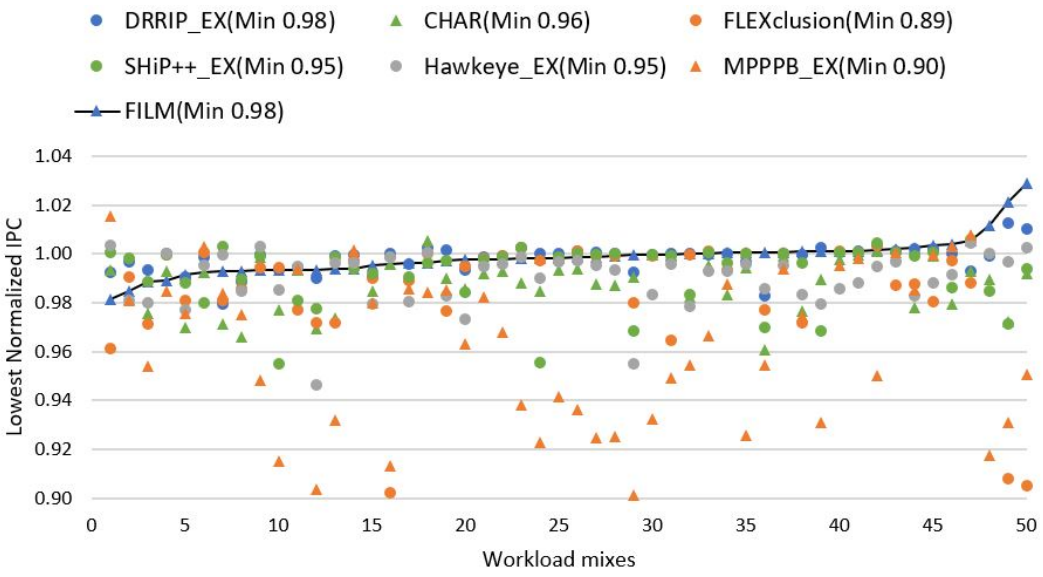


Figure 5.8: Lowest normalized IPC of any co-running program. IPC reduction due to negative interference is least for FILM.

ness metric that captures any performance degradation ($speedup < 1.0$) of a co-running workload in a mix. FILM is able to restrict the performance degradation of single application within 2%, with the highest minimal speedup of 3%. The minimal speedup of other policies ranges from 0.90 to 0.96 (i.e., performance degradation of 10% to 4%).

5.2.1.4 Comparison with RAP

As both FILM and RAP [123] have the same goal of optimizing data placement across the cache hierarchy, the evaluation of FILM scheme is completed in this section with comparison with RAP. RAP is compared separately because it does not have special training mechanism for prefetched blocks. With prefetching disabled, FILM's performance on the multi-programmed

Table 5.4: Comparison between FILM and RAP enhanced with FILM-like training on prefetch relative to TC-UC

Metric	Enhanced RAP	FILM
Hardware overhead per core	72KB	8.5KB
Energy efficiency (IPC/J)	0.96	1
DRAM traffic (access count)	1.06	1
LLC traffic (access count)	1	1
Performance (IPC)	0.98	1

mixes beats RAP by 7%. With prefetching enabled, the performance of the original version of RAP on the system with prefetching enabled is poor, because RAP uses PC as its training metric and the PC of a prefetched block is zero (prefetcher does not have PC). To make a fair comparison, RAP is enhanced with FILM-like training schemes on prefetch. The result of the enhanced RAP is shown in Table 5.4. One difference between RAP and FILM is that for cachelines with frequent accesses(hits), RAP learns from only the first hit and evictions, whereas FILM learns from all the hits and evictions. Another difference is that to avoid losing a global view of data movement under heavily data bypassing, RAP dedicates few sets as learning sets which are not affected by the RAP bypass algorithm, whereas FILM follows its "utilize empty line" rule. Enhanced RAP has a 72KB hardware cost as it extends the metadata field of each cacheline in the cache subsystem with 12-bit PC. With such significant overhead, the energy efficiency of Enhanced RAP is 4% less than FILM. Both Enhanced RAP and FILM cut down LLC traffic due to their support on cache level bypassing. RAP has 6% more DRAM accesses

Table 5.5: FILM hardware budget (per core)

Component	Parameter	Budget
Prediction Learning table (Demand + Prefetch)	16 entries, 11-bit Tag, 2048-bit footprint container 8-bit ReuseCnt+Fill	4 KB
Result table	2048 entries, 2-bit entry	0.5KB

Table 5.6: Overhead Comparison (per core)

CHAR	SHiP++_EX	Hawkeye_EX	MPPPB_EX	FILM
2.25KB	77.5KB	86KB	97KB	4.5KB

and 2% less performance compared with FILM as its bypassing tends to get overly aggressive.

5.2.2 Hardware cost and design decisions

5.2.2.1 Hardware overhead comparison

Table 5.5 shows the hardware budget of FILM's two main memory components, Prediction Learning table and Result table. FILM's total hardware budget is 8.5KB per core, which is 0.3% more SRAM than the three-level cache hierarchy, in exchange of significant reduction in data movement and dramatic improvement on energy efficiency. Table 5.6 compares the hardware budgets for the evaluated replacement policies. TC-UC and DRRIP_EX are not listed because they, as well as other six schemes, share the common overhead of three bits per cacheline to store re-reference counter. FLEXclusion is not listed because it leverages pre-existing data paths with only four registers overhead. SHiP++_EX, Hawkeye_EX and MPPPB_EX have 72KB more overhead than

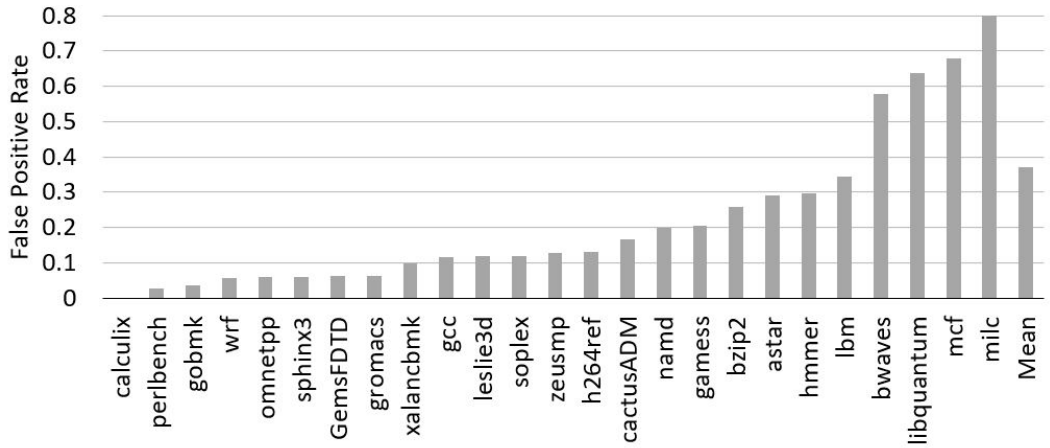


Figure 5.9: Rate of multiple entry matches reported by FILM's bloom filter.

the number claimed in their paper due to the overhead of 14-bit PC-based signature stored with cacheline.

5.2.2.2 Bloom filter analysis

As mentioned earlier, a bloom filter is used in FILM, to tie all cache blocks fetched from DRAM by the same memory instruction to one table entry. When training, this bloom filter is used to identify which specific memory instruction to train. FILM stops training on a data address when a multiple match is detected. Figure 5.9 illustrates the false positive rate at which FILM's bloom filter reports multiple entries matched one address among SPEC CPU2006 workloads. The false positive rate is greater than 10% for more than half of the workloads. One may be surprised that FILM's training accuracy is not seriously impacted even with such high rates of multiple matches. For workloads like libquantum and bwaves, the valid training points are only

around 30% of the entire training set, and this observation suggests high information redundancy in the training set. In other words, although FILM is not trained based on the entire history of data accesses generated by a given instruction, a small portion of the history provides sufficient information to train a decision that is as good as the one made with all the history. Further, this observation adds confidence to the design choice of using a few sampled cache sets for training FILM, as opposed to tracking all the sets.

Figure 5.10 illustrates the impact on performance as the number of bloom filters in the Prediction Learning table increases from 8 to 64. The performance number is normalised to 16 entries, which is the same number showing in Table 5.5. A huge performance jump is seen when the number of bloom filters increases from 8 to 16. Performance difference between 16 and 32 bloom filters are negligible. Performance increase by 2% as the number quadruples from 16 to 64. Thus, the least amount to use to maintain performance is 16.

5.2.2.3 Impact of "Utilize empty blocks" rule

The "Utilize empty blocks" rule inserts evicted cache blocks into lower caches when there is empty space in the cache, regardless of FILM's bypassing hints. If such blocks see hits in their new home, it guides FILM to dynamically adjust its outdated bypass decisions. The CHAR algorithm uses empty block as well. However, FILM uses empty blocks to create opportunities for detecting stale bypassing hints, whereas CHAR does not perform any special

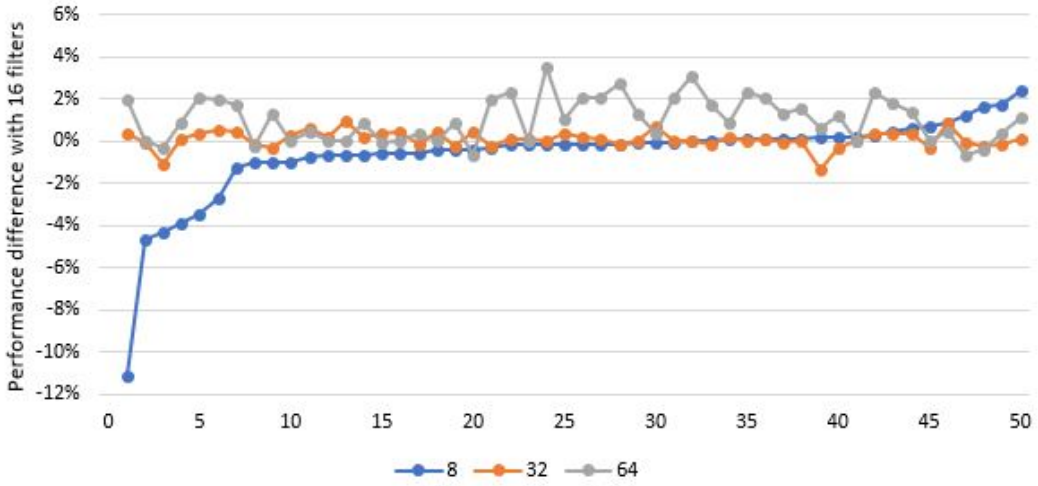


Figure 5.10: Performance sensitivity to the number of bloom filters. IPC normalized to 16 bloom filters.

training on data inserted into empty blocks. Figure 5.11 compares normalized energy efficiency over TC-UC between FILM and another FILM implementation which does not apply the "Utilize empty blocks" rule and always bypasses data block based on hints. Always bypassing reduces the number of LLC installs and saves cache energy, at the cost of losing performance and increasing DRAM energy. Figure 5.11 illustrates that "utilize empty block" rule contributes to an average of 4% energy efficiency improvement compared to TC-UC.

5.3 Summary

Due to the inherent difference of data block insertion and movement between an exclusive hierarchy and an inclusive hierarchy, prior work, which is PC-correlated and is designed with inclusive caches in mind, cannot be

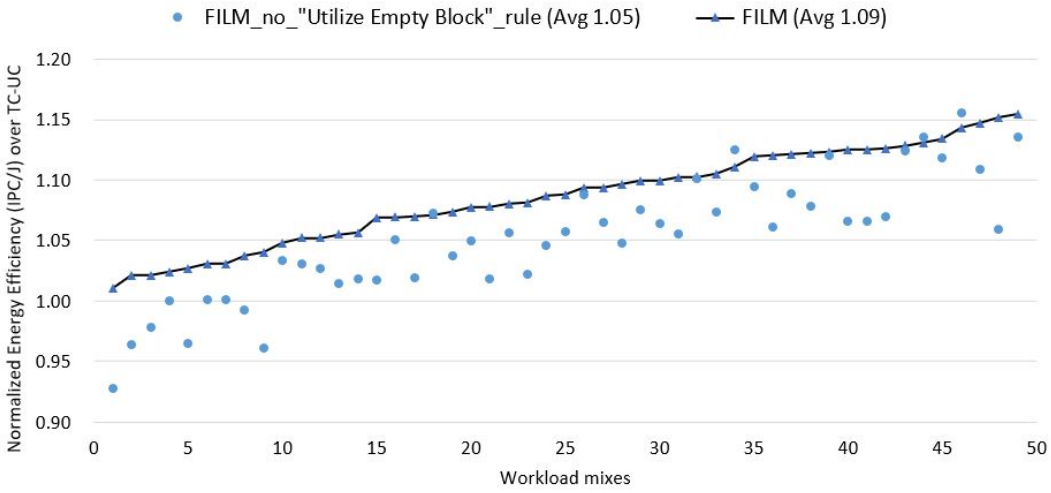


Figure 5.11: Energy efficiency of FILM and FILM without "utilize empty block rule". Results normalized to TC-UC. The higher the better.

easily applied to exclusive caches. Moreover, a holistic approach to manage data placement is essential for high cache performance and efficient resource utilization. Therefore, FILM, a locality filtering mechanism is proposed to adaptively guide data placement into appropriate cache layers based on data reuse patterns. With a PC-based prediction scheme, FILM utilizes bloom filters to record the memory instruction PC of data blocks, incurring minimal cache overhead for meta-data transmission and storage. Besides, FILM is able to quickly detect and correct any stale bypass decisions. FILM also does special training on prefetch requests, and makes prefetch aware learning of bypass/placement decisions.

Compared to a competitive baseline (TC-UC), FILM improves the average energy efficiency of multicore multi-programmed system by an of average 9% (maximum 20%), beating the second highest average energy efficiency im-

provement from CHAR by 5%, and is constantly more energy efficient than other PC-correlated schemes. Moreover, FILM cuts down wasteful cache block insertions and data movement, and generates on average 12% less LLC traffic and 4% less DRAM traffic than other PC-correlated schemes.

Chapter 6

GPU Data Placement Optimization

GPU has traditionally been an accelerator for graphics processing, but recently has seen large adoption as a general high-performance computing device. With the SIMT execution model and massive multithreading, GPU provides incredible speedups for embarrassingly data parallel applications. However, compared to the hundreds or thousands of threads running simultaneously on the GPU, GPU cache capacity of few mega-bytes is too small and resource contention between threads is extremely severe. The cache capacity per GPU thread and the data lifetime in GPU caches are both smaller compared to CPUs. Cache management schemes proposed for CPUs can not fully exploit the utilization of GPU memory subsystem. One solution to solve the memory bottleneck of GPU applications is through explicitly managing data placement using scratchpad memory. With the help of scratchpad RAM, software programmers and compilers can encode data placement hints in the application code to directly inform GPU memory subsystem about future data access pattern, such that only data blocks with future reuse are stored on chip to reduce off-chip data re-accesses. In this chapter[1], a popular high perfor-

[1]Contents of this chapter was previously published at the International Conference on Parallel Processing Workshops in 2016 [138]. I am the principle author of this work.

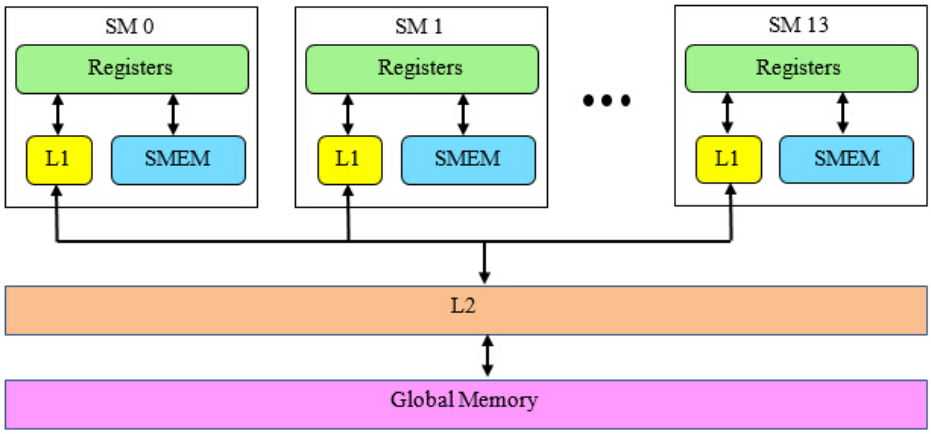


Figure 6.1: GPGPU memory hierarchy

mance computing application, kernel summation, is selected as an example to illustrate the performance and energy benefit of managing data placement using scratchpad memory in GPU.

6.1 GPGPU background

The GPGPU architecture studied in this dissertation is the NVIDIA Maxwell architecture. It is composed of a set of compute units, a cooperative thread array (CTA) scheduler, a unified L2 cache, and global memory. Each compute unit, also called "Streaming Multiprocessor" (SM), contains a number of arithmetic and logic units, a large register file, a shared memory, non-coherent caches, and a scheduler for units of execution. The units of execution are referred to as warps and each warp is composed of 32 scalar threads. All threads within a warp are scheduled together, and thus are implicitly

synchronized. Those threads can exchange values using either shared memory or the shuffle instruction.

A CTA or a thread-block is a group of warps that execute concurrently on an SM. Threads executing on the same SM share a shared memory and are explicitly synchronized using barriers or memory fences. The CTA scheduler only allows a CTA to execute on an SM once the amount of required shared memory and registers are available. Having a large number of CTAs and scheduling warps concurrently on an SM allow the hardware to hide memory latency. When a warp experiences a long memory stall due to cache misses, bank conflicts or un-coalesced accesses, GPGPUs can hide latency by bringing in other warps to concurrently execute compute instructions.

The memory hierarchy of the Maxwell architecture is shown in Figure 6.1. Each SM contains separate shared memory (SMEM) and unified L1 cache. Memory accesses of all SMs must go through a shared L2 cache. The shared memory is a programmer managed cache, which is usually used in conjunction with barriers, to communicate values between threads in a CTA. Unlike the NVIDIA Fermi architecture [3], shared memory becomes an individual unit in the Maxwell architecture and L1 cache is unified with texture cache [7]. By default, the unified L1 and texture unit of the Maxwell architecture does not actually cache global loads, except for gather instructions, texture fetches, and surface writes. However, a compiler flag can be used to specify that all global loads must be cached at all levels.

The design of shared memory requires that shared memory is as large as

possible and provides bandwidth high enough to service 32 threads per cycle. In order to provide high bandwidth, shared memory is laid out as a series of banks (32 for the Maxwell architecture), where each bank is four bytes wide. The width of bank is chosen to be the same as the size of float and *RGBA* data types, which are frequently used in graphics applications. Bank conflict occurs when different words in the same shared memory bank are accessed by threads in the same warp. NVIDIA makes it known that good performance from shared memory cannot tolerate bank conflicts. Programmers are encouraged to use memory access patterns that do not cause a bank conflict. All threads in a warp can issue a shared memory load in the same cycle. It is seen that all 32 banks share the same row select, which means that in order to avoid bank conflicts, in addition to using different banks, threads need to access memory within the same 128-byte region. If there is no bank conflict, 32 requests turn out to be one shared memory transaction and exploit the high bandwidth of the shared memory. The register file is also banked. Register bank conflicts are usually avoided with the help of compiler, and are only likely to occur when storing large vectors in registers.

6.2 Kernel Summation application

Kernel summation is a technique used to approximate the interactions between two sets of points in a high dimensional space. It is widely used in data analysis, electrostatics, and particle physics, most famously N-body simulations. There are numerous studies that have proposed scalable algorithms

and high-performance implementations of fast kernel summation schemes such as treecodes [94][16], fast multipole methods [22][73], particle-mesh methods [120], Ewald sums [30], etc. These algorithms can scale to billions or trillions of points for problems in two or three dimensions. However, they do not scale to higher values of K because they depend linearly or super-linearly on K. Other algorithms which are efficient for high dimension K apply GEMM defined in BLAS library [23]. Given $\alpha_i, \beta_j \in \mathbb{R}^K$ from a set of source points and target points, a **kernel** $\mathbb{K}(\alpha_i, \beta_j)$ describes the pairwise interaction between two points. In this dissertation, K is denoted as the dimension of the space. Gaussian kernel is selected as an example and is defined as

$$\mathbb{K}(\alpha_i, \beta_j) = \exp^{-\frac{\|\alpha_i - \beta_j\|_2^2}{2h^2}} \tag{6.1}$$

where h is a constant. Solving the **kernel summation** problem is to compute a scalar value V_j (associated with the target point β_j) such that

$$\mathrm{V_j} = \sum_{i=1}^{N} \mathbb{K}(\alpha_i, \beta_j) W_i \tag{6.2}$$

where $V \in \mathbb{R}^N$ is an N dimensional potential vector, and $W \in \mathbb{R}^N$ is an N dimensional weight vector. This dissertation focused on accelerating the evaluation of the Equation 6.2, for modest N (O(10,000)).

Solving kernel summation problem requires computing the pairwise interaction between every source point $\alpha_i \in \mathbb{R}^N$ and every target point $\beta_j \in \mathbb{R}^N$. An efficient way to compute the N^2 interactions is to use the following identity for the Euclidean distance between two K dimensional points α_i and β_j:

$$\|\alpha_\mathrm{i} - \beta_\mathrm{j}\|_2^2 = \|\alpha_i\|_2^2 + \|\beta_j\|_2^2 - 2\alpha_i^T \beta_j \tag{6.3}$$

An obvious way of evaluating all the pairwise interaction is to treat the source point set and the target point set as two K by N input matrices and to apply GEMM to compute the $-2\alpha_i^T\beta_j$ component for all $i \in N$ and $j \in N$. Each elements of the GEMM output matrix will be added to the remaining components, $\|\alpha_i\|_2^2 + \|\beta_j\|_2^2$. By performing an exponential function (Equation 6.1) on the output matrix of the previous step, a temporal result matrix $\mathbb{K}$ is generated with each element located at (row i, column j) represents the pairwise kernel interaction between the source point α_i and the target point β_j. In the end according to Equation 6.2, a GEMV is applied to the matrix $\mathbb{K}$ and the weight vector W to get the result vector V.

A simplified kernel summation algorithm is shown in Algorithm 1. Inputs A and B are M-by-K and K-by-N matrices separately, and W is an N-dimensional weight vector. α_i is a K-dimensional row vector representation and β_j is a K-dimensional column vector representation.

6.3 Proposed Scheme

Using vendor-provided libraries brings performance benefits through the highly optimized BLAS, but it also sacrifices data locality because the intermediate matrix, as the return value of GEMM call, is written back to main memory due to its huge size not fitting into caches. The kernel summation problem typically involves large data sets, and the long memory access latency is the crucial bottleneck of program execution. Except from losing data locality, energy and power spent in memory access is another factor urg-

Algorithm 1 Basic kernel summation

Inputs:
$A = [\alpha_0, \alpha_1, ..., \alpha_{M-1}]^T$, M-by-K matrix
$B = [\beta_0, \beta_1, ..., \beta_{N-1}]$, K-by-N matrix
$W = [\omega_0, \omega_1, ..., \omega_{N-1}]^T$, N-by-1 vector
Outputs:
$V = [\nu_0, \nu_1, ..., \nu_{N-1}]^T$, N-by-1 vector
$vec\alpha \leftarrow [\|\alpha_0\|_2^2, \|\alpha_1\|_2^2, ..., \|\alpha_{M-1}\|_2^2]^T$
$vec\beta \leftarrow [\|\beta_0\|_2^2, \|\beta_1\|_2^2, ..., \|\beta_{N-1}\|_2^2]$
// duplicating $vec\alpha$ N times to form a M-by-N matrix
$squareA \leftarrow [vec\alpha, vec\alpha, ..., vec\alpha]$
// duplicating $vec\beta$ M times to form a M-by-N matrix
$squareB \leftarrow [vec\beta, vec\beta, ..., vec\beta]^T$
// GEMM
$C \leftarrow A \times B$
$R \leftarrow squareA + squareB - 2 \times C$
for each element in R **do**
    $K(i,j) \leftarrow exp\{-\frac{R(i,j)}{2h^2}\}$
end for
// GEMV
$V \leftarrow K \times W$

ing a better solution. The increased power and energy consumption and the resulting thermal issues have become major challenges. Memory, or DRAM, operations usually take 20%-40% share of total energy consumption in many applications. DRAM energy has been reported to account for 22% in UltraSPARC T1 systems [48], more than 25% of data centers [92], and around 40% in a mid-range IBM eServer machine [77].

To address the performance and energy challenges, fused kernel sum-

mation is proposed. The steps of kernel summation are fused into the GEMM structure since most of the redundant memory accesses are coming from GEMM. Fusion enables consumer operations to access data directly from registers and caches right after producer completes, and thus increases data locality and relieves memory burden. The problem is decomposed into parallel tasks with minimal communication and synchronization, and assign each task to one GPU thread block. The GEMM part of each task is fully parallel. When a thread block completes the GEMM portion, it could go on to use intermediate value, i.e. the GEMM output, stored in registers or shared memory to perform kernel evaluation without waiting for other thread blocks. The only data that a thread block stores back to main memory is the partial sum of the final result. Communication between thread blocks happens only in the GEMV part where all thread block outputs are accumulated to calculate a final result. Instead of waiting for all the thread blocks to be ready before aggregating, the reduction operation is done through each thread block accumulating its output to the latest reduction result in an atomic way. In other words, a thread block immediately retires after updating the final result with its own output, and only one thread block is allowed to update the final result at any time. The problem size of these tasks and the size of thread blocks are selected to strike a balance between higher device occupancy and less data duplication, which infers less memory accesses.

6.3.1 Data placement in GEMM

The algorithmic view of my SGEMM is shown in Figure 6.2. Matrices A and B hold coordinate table of the source point set and the target point set respectively. For the remainder of this chapter, M denotes the leading dimension of matrix A and K denotes the leading dimension of matrix B. So the matrix A is of size M by K, and the matrix B is of size K by N. It is assumed that the matrix A is in row major order and the matrix B is in column major order. As shown in the figure, all the three matrices are divided into sub-matrices. $C_{i,j}$ denotes a $submatrixC$ which has i submatrices to its left and j submatrices on its top; A_i denotes a $submatrixA$ which has i submatrices on its top; and B_i denotes a $submatrixB$ which has i submatrices to its left. A thread block with a block ID (bx, by) is assigned to compute $C_{bx,by} = A_{by} \times B_{bx}$, and all thread blocks can be executed concurrently without race conditions. $submatrixA$ and $submatrixB$ are partitioned into tiles of size 128 by 8 and 8 by 128 separately. A thread block performs rank-8 update across the K dimension, i.e.,

$$\text{submatrixC} = \sum_{\text{i=0}}^{\text{K/8}} \text{tileA}_\text{i} \times \text{tileB}_\text{i}$$

A $submatrixC$ is divided further into 16x16 microtiles. $microtileC_{i,j}$ denotes an 8 by 8 microtile which has i microtiles to its left and j microtiles on its top within the range of a $submatrixC$. Threads are organized into a 16 by 16 grid to form a thread block, and the thread with threadID (tx, ty) is corresponding to the $microtileC_{tx,ty}$. Therefore the task of a thread block computing

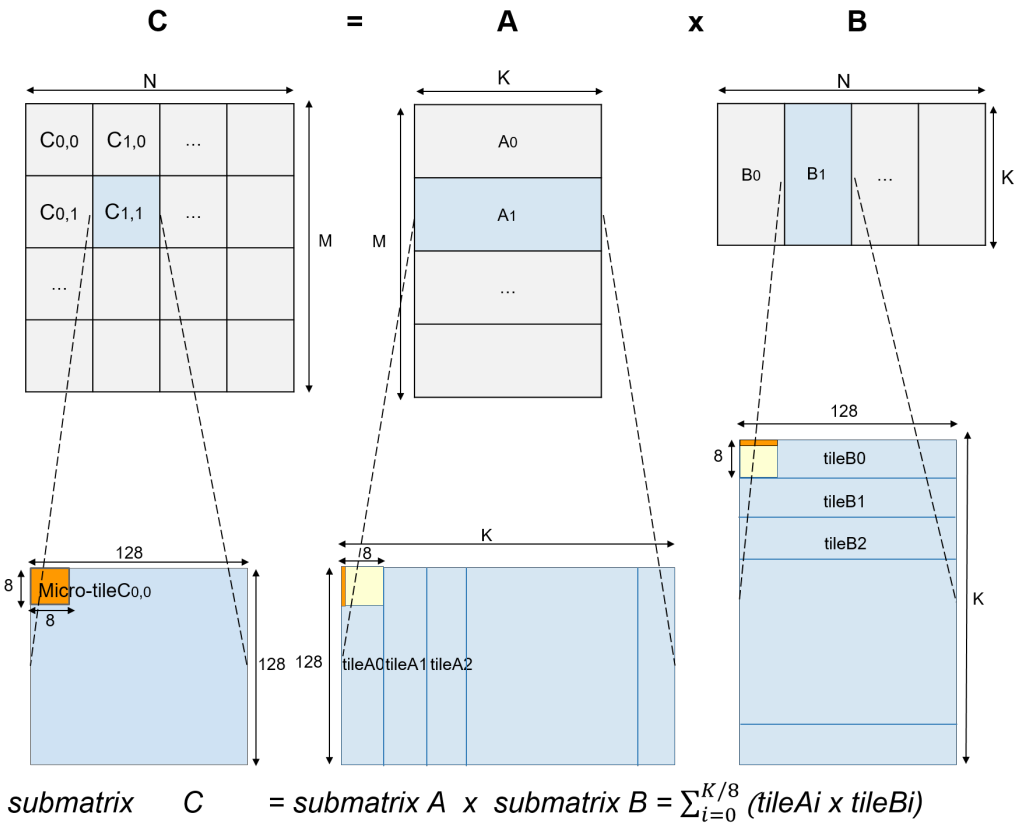


Figure 6.2: GEMM algorithmic view

$tileA \times tileB$ is decomposed into each thread computing $microtileC_{tx,ty} = microtileA_{ty} \times microtileB_{tx}$

Based on the data access pattern of each thread block, a $submatrixA$ or a $submatrixB$ will be accessed multiple time by different thread blocks. Taking a $submatrixA$ of size m by K as an example, it would be accessed by N/m thread blocks, which indicates that the entire matrix A is repeatedly loaded N/m times. Even though the shared L2 cache would serve data reuse

among thread blocks, it depends on the thread block scheduling policy to ensure that thread blocks accessing the same range of memory are activated at the same time. Besides, considering the limited size of L2 and the large matrix size, average L2 size per thread block is not large enough to reduce repeated memory accesses. Theoretically speaking, the value of m, which is the leading dimension of both $submatrixA$ and $submatrixC$, should be sufficiently large to to reduce the N/m reloading times of matrix A. In other words, the partition of matrix A and matrix B should be relatively coarse grained in order to reduce reloading times, which directly influences the size of $submatrixC$.

Factors like GPU limits, trade-offs between high SM occupancy and less data locality, inter-influence between matrix size and matrix partition are taken into consideration when determining the size of $submatrixC$ and decomposing $submatrixC$ computation to thread level tasks. In the best scenario of experiments, a thread block of dimension 16 by 16 computes a $submatrixC$ of 128 by 128, and each thread computes 8 by 8 elements.

The number of physical registers is one of the performance bottlenecks of my solution. Programmers have the ability to choose how much shared memory space is consumed using the CUDA C programming language, but they can not explicitly control register usage without support from assembly, which is not yet released by NVIDIA. The test machine, GTX970, provides an upper bound of 65536 registers per SM. In other words, up to 255 registers can be allocated to each thread when the thread block dimension is 16 by 16.

If nothing is limiting performance, larger number of registers used per thread would lead to lower SM occupancy. In this solution, each thread takes 64 registers to hold 64 partial sums of $microtileC$ in order to achieve the best data locality. Each thread performs a rank-1 update to maximize the computation to load ratio, hence vector operands from $tileA$ and $tileB$ take another 16 registers. Including miscellaneous essential demands like thread index and control flow variables, 96 to 128 registers are consumed by each thread and this leads to having up to two thread blocks executed simultaneously in the same SM. Although the compiler option of "*–maxregcount*" helps achieve higher occupancy, register spilling has a huge negative impact on performance because of additional L1 transactions due to spilling.

In this implementation, partial sums of $submatrixC$ are stored in the thread register file, and $tileA$ and $tileB$ are loaded into the shared memory sequentially. Double buffering is used to hide shared memory load latency. Double buffering requires size of tiles to be restricted in such a way that shared memory can hold at least two pairs of tiles at any moment. When one pair of ($tileA_i$, $tileB_i$) are used in computation, next pair of ($tileA_{i+1}$, $tileB_{i+1}$) could be loaded into shared memory. In the Maxwell assembly, each load is marked by an integer. Explicit synchronization is inserted to guarantee that loading a pair of tiles completes before being consumed in the next computation step.

In this solution, up to two thread blocks could be executed simultaneously in the same SM. Each thread computing more than 8x8 C elements will reduce the occupancy to one thread block per SM due to the register

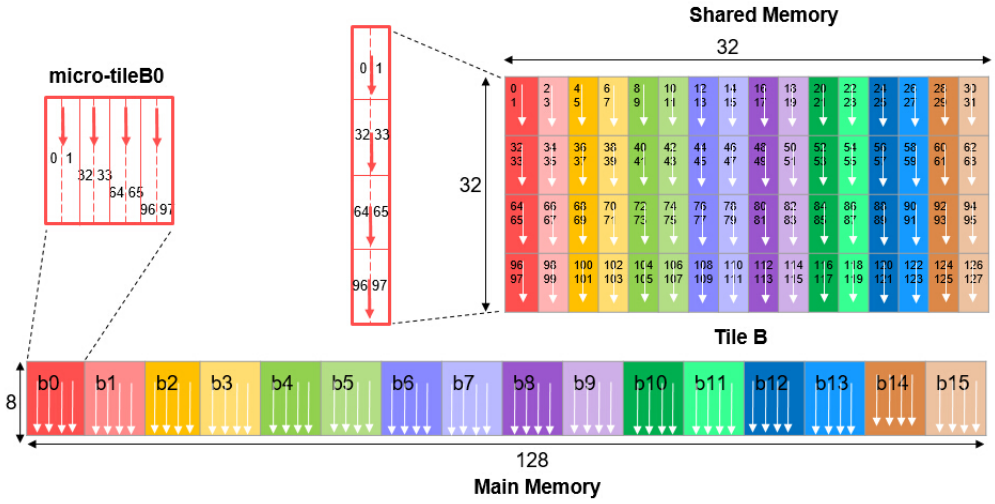


Figure 6.3: Data-thread mapping when loading *tileB* into shared memory

count limit. On the contrary, computing fewer C elements will transfer the bottleneck to other parts. For example, if 128x128 elements of *submatrixC* are computed by one thread block and 4x4 C elements per thread, it would then require 1024 threads per block. Occupancy is still two thread blocks per SM due to the device limit of 2048 threads per SM.

6.3.2 Shared memory data mapping

The shared memory in GPU serves like a scratchpad. Programmer is directly responsible for all shared memory accesses. The shared memory performance is a combined effect of the number of bank conflicts, the granularity of access, and the total number of accesses. Larger granularity of access means less load instructions and higher bus bandwidth utilization. For example, loading four float values in one load instruction in the *float4* data type rather than

four load instructions in the float type results in fewer load instructions. One important consideration in using shared memory is to avoid bank conflicts. When a bank conflict occurs, shared memory instructions are required to be replayed for certain threads. There won't be any shared memory bank conflicts when all threads in the same warp access the same data, because shared memory does have some broadcast capabilities. For instance, if all 32 threads access the same four bytes in a single bank, all requests can be serviced in a single cycle. The broadcast capability also extends beyond a single broadcast, such as the same value requested by eight threads within the same warp would be served in one broadcast within single cycle.

When loading $tileA$ and $tileB$ into shared memory, one half of the thread block (128 threads) loads $tileA$ and the other half loads $tileB$. Threads are carefully mapped to elements in memory to avoid shared memory store bank conflicts. Figure 6.3 illustrates how to avoid both load and store bank conflicts when bringing $tileB$ into shared memory. Loading and placing $tileA$ is similar to $tileB$, and $tileA$ is also divided similarly into 16 eight by eight microtiles. The numbers illustrated on the figure are linear thread index and they tell which thread is accessing that part of the main memory or writing that bank of shared memory. A tile in main memory is partitioned into 16 eight by eight microtiles, and each microtile is further divided into eight eight by one tracks, each of which is accessed by a different thread. In the shared memory, $tileB$ is stored in a two-dimensional array data structure in which each row consists of 32 elements sitting in the 32 banks of shared memory.

Intuitively, each of the all 32 threads within a warp would load a track from a group of eight neighboring microtiles, and store the track to one bank of shared memory. However such track placement would not solve the problem of load bank conflicts. Since thread with threadId (tx, ty) will touch $microtileA_{ty}$ and $microtileB_tx$ during multiplication, a warp needs to load all 16 microtiles of $tileB$, and evenly spread them among 32 banks to get rid of load bank conflicts. Therefore, the placement of $tileB$ in shared memory need to be re-arranged to avoid load bank conflicts, and the match between a thread and a track is not that intuitive.

As shown in the figure, in order to spread 16 microtiles among 32 banks, an eight by eight microtile in main memory is reconstructed as 32 by two. A warp loads 32 tracks from main memory by picking two tracks per microtile, and places them side by side in shared memory. It takes collaboration of four warps to store one microtile. For example, $microtileB_0$ is divided into four groups of tracks. Thread 0, 1 in warp 0 will store data of group 0 to location (bank 0-1, row 0-7); and thread 32, 33 belonging to warp 1 will write group 1 tracks into location (bank0-1, row 8-15), and so on. This guarantees that the 32 threads in the same warp are writing to 32 different banks in shared memory and no load bank conflicts would occur. Generally speaking, a thread will touch track $[tx \bmod 2 + 2 \times (ty \bmod \frac{blockDim.y}{2})]$ of $microtileB\lfloor\frac{tx}{2}\rfloor$, and store the track into bank$[tx \bmod 32]$, row (8ty to 8ty+7).

Algorithm 2 Fused kernel summation pseudo code for each thread block

Inputs:
matrix $subA$(128 by K), $subB$(K by 128), $subA2$(128 by 128),
$subB2$(128 by 128).
vector $subW$(128 by 1), $subV$(128 by 1). blockId(bx, by)
Outputs:
vector $partialV$ (128 by 1)
Initialize:
$j \leftarrow 0, i \leftarrow 0$, declare $sharedA_0$, $sharedB_0$, $sharedA_1$ and $sharedB_1$
as arrays in shared memory
temporal matrix T (128 by 8), $T = [\tau_0, \tau_1, ..., \tau_{127}]^T$, τ is an 8 by 1
row vector,
γ is T's 8-dimensional column vector,
$\gamma_{i,j} = [T[8i, j], T[8i + 1, j], ..., T[8i + 7, j]]^T$
parfor each thread with threadId (tx, ty) **do**
  $load\text{-}nonblocking(sharedA_j \leftarrow tileA_i, sharedB_j \leftarrow tileB_i, tx, ty)$
  syncthreads();
  for i from 1 to $\frac{K}{8} - 1$ **do** // GEMM. $subC = subA \times subB$
    $j \leftarrow j \oplus 1$ // $\oplus$ is Exclusive OR operator
    $load\text{-}nonblocking(sharedA_j, sharedB_j, tileA_i, tileB_i, tx, ty)$ // Memory access
    $microtileC_{tx,ty} += microtileA_{ty} \times microtileB_{tx}$ // Hide memory access latency with Computation
    syncthreads()
  end for
  $microtileC_{tx,ty} += microtileA_{ty} \times microtileB_{tx}$
  $subC[tx, ty] \leftarrow \exp\{-\frac{subA2[tx,ty]+subB2[tx,ty]-2\times subC[tx,ty]}{2h^2}\}$ // Gaussian Kernel Evaluation
  // Summation
  $\gamma_{tx,ty} \leftarrow microtileC_{tx,ty} \times subW_{tx}$ // Intra-thread level reduction.
  syncthreads()
  if $ty \leq \frac{blockDim.y}{2}$ **then**
    $tid \leftarrow ty \times blockDim.x + tx$
    $partialV[tid] \leftarrow rowReduction(\tau_{tid})$ // Intra thread block level reduction
    atomicAdd($subV[tid], partialV[tid]$) // Inter thread block level reduction
  end if
end parfor

6.3.3 Kernel summation fused with GEMM

Steps of kernel summation are fused into the GEMM framework as described above. The pseudo code shown in the Algorithm 2 demonstrates an

overview of the fused kernel summation routine executed by each thread block. There are seven inputs: $subA$ and $subB$ are 128 by K and K by 128 matrices separately; $subA2$ and $subB2$ are 128 by 128 matrices, which are submatrices of the $squareA$ and $squareB$ computed in the Algorithm 1; $subW$ is part of the weight vector W; $subV$ frames the final result V of the kernel summation problem; and two-dimensional thread block index (bx, by). The output of each thread block is vector $partialV$. A representation of (tx, ty) refers to the two-dimensional thread index. The same partitioning scheme of matrices A, B, and C from the previous part is followed. Additionally $squareA$ and $squareB$ are divided into sub-matrices the same way as C and the same denotation rule. Both vector W and V are evenly split into sub-vectors of dimension $blockDim.y$. The variables $sharedA_0, sharedA_1, sharedB_0, sharedB_1$ and T are declared per thread block and their sizes are the same as tileA and tileB. Matrix T is used to store thread level reduction result in shared memory. In my implementation code, T explicitly reuses the shared memory spaces of $sharedA_0$ in order to limit the amount of shared memory resources required per thread block and to increase SM occupancy. Denotation $X[i, j]$ represents the element in the i-th column and j-th row of matrix X, and $Y[k]$ represents the k-th element of vector Y. The mapping of thread blocks to the first element of their input matrices and input vectors are shown below. For example, when matrix A and B are partitioned into blocks and indexed in the same way as the one shown in Figure 6.2, a thread block whose index is (bx,by) will load the by-th block of matrix A as its program input subA and bx-th block of

matrix B as subB.

$$\text{subA} = A + 128 \times by$$

$$\text{subB} = B + 128 \times bx$$

$$\text{subA2} = squareA + N \times by + 128 \times bx$$

$$\text{subB2} = squareB + N \times by + 128 \times bx$$

$$\text{subW} = W + 128 \times by$$

$$\text{subV} = V + 128 \times by$$

Line 5-13 in Algorithm 2 are the same GEMM structure as I described in the previous part. In the function load-nonblocking ($sharedA_j$, $sharedB_j$, $tileA_i$, $tileB_i$, tx, ty), each thread in the first half of thread block (i.e. $ty \leq \frac{blockDim.y}{2}$) would load a track from the $tileA_i$ into the shared memory variable $sharedA_j$, and each thread in the other half would load a track from the $tileB_i$ into the shared memory variable $sharedB_j$. The mapping of threads to tracks and data placement are already discussed before. At the end of the GEMM routine, each thread completes updating a $microtileC$, and this intermediate product is held in thread registers. The kernel evaluation according to Equation 6.1 becomes embarrassingly parallel. In order to make full use of the benefit brought by register locality, each thread performs kernel evaluation in the next step (line 14).

There are three levels of reductions during kernel summation: intra-thread level, intra-thread-block level, and inter-thread-block level. Synchronization needs to be carefully taken care of in the last two levels. During the

Table 6.1: Configuration

Number of Multiprocessors	13
Maximum number of threads per block	1024
Warp size	32
Maximum number of resident threads per multiprocessor	2048
Number of 32-bit registers per multiprocessor	64K
Maximum number of 32-bit registers per thread	255
Maximum amount of shared memory per multiprocessor	96KB
Shared Memory Bank Size	4B
Number of shared memory banks	32
Number of warp schedulers	4
L2 size	1.75MB

intra-thread level summation, each thread performs row reduction on its eight by eight microtile, and stores the result in shared memory. The intra-thread-block level reduction needs to wait until all threads complete its own reduction work. A thread block level synchronization function, *syncthreads()*, is called to ensure function correctness. In my case since there are 16 threads in the x-dimension of thread block, intra-thread-block level summation reduces results of every 16 threads in the row to form a partial result of that thread block. Because there are 128 rows in $subC$, only half of the thread block (i.e. 128 threads) is required to perform intra-thread-block reduction, with each thread responsible for one row. Notice that the output $partialV$ of each thread block is not the subvector of final result V. Instead $subV$ is the sum of $partialV$ distributed across thread blocks with the same by. Data communication between thread blocks is done through main memory, and requires waiting for all thread blocks to finish execution. In order to avoid synchronization latency

between thread blocks and to prevent accessing memory twice to store and reload $partialV$, an atomic add operation is chosen to update $subV$ whenever $partialV$ is ready.

6.4 Evaluation

The kernel summation application is run on a desktop system equipped with a Corei5-4960K connected to an NVIDIA GTX970 Maxwell GPU (4GB of GDDR5 video memory) over a PCIe interconnect. Technical specifications of the GTX970 are listed in Table 6.1, and all numbers are based on the latest compute capability of 5.2. All the performance metrics and events in this work are measured with the nvprof [6] profiling tool provided by NVIDIA. The cuBLAS library used in this work is version 7.0.

Three different implementation of kernel summation problem are run and compared, which are denoted ***Fused***, ***CUDA-Unfused*** and ***cuBLAS-Unfused***. *Fused* is the kernel fusion implementation discussed in the previous section. Two unfused versions of kernel summation were programmed. One is to pair self-implemented SGEMM with the kernel evaluation and the summation routine, denoted by *CUDA-Unfused*. Another one is to call the SGEMM function provided in the cuBLAS library followed by the kernel evaluation and the summation routine, denoted by *cuBLAS-Unfused*. All runs are repeated multiple times to ensure the repeatability and consistency of the results. The different kernel summation solution were tested and compared with four groups of different K value, which are 32, 64, 128, and 256.

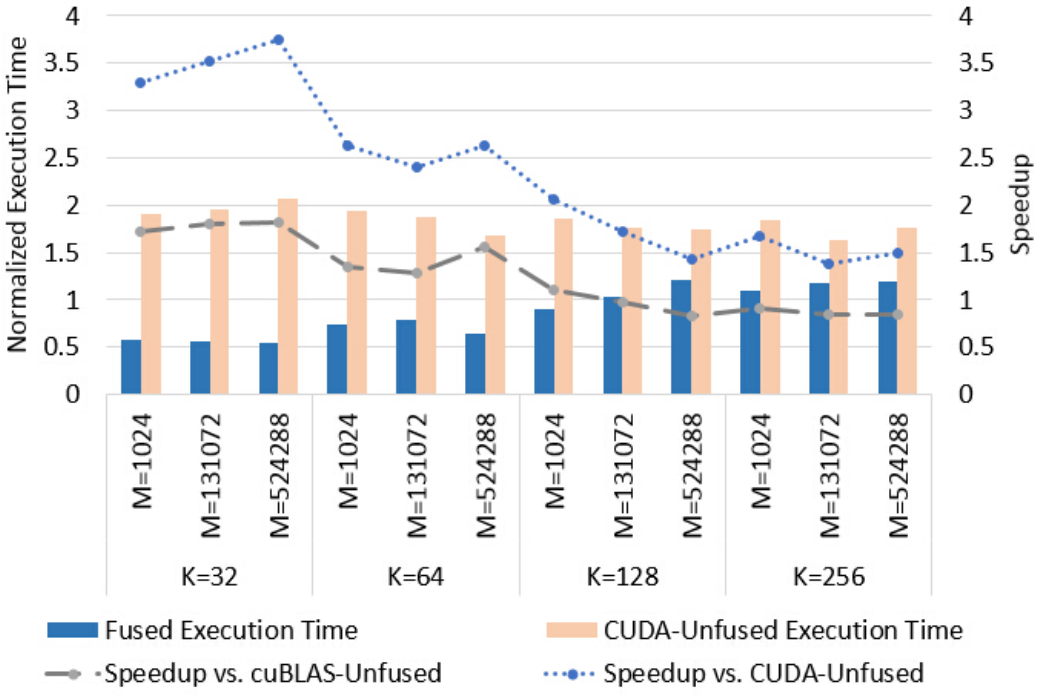


Figure 6.4: Execution time and speedup of the fused kernel summation in comparison with unfused implementations.

The advantage of fused kernel summation is seen from both performance and energy perspectives. Energy model is built based on GPUWattch.

Figure 6.4 demonstrates normalized execution time of the *Fused* and the *CUDA-Unfused* implementations with respect to the *cuBLAS-Unfused* implementation on primary axis, as well as the speedup of the *Fused* kernel summation versus both *cuBLAS-Unfused* and *CUDA-Unfused* on secondary axis. *Fused* approach beats *cuBLAS-Unfused* approach by up to 1.8X speedup when dimension K is not extremely large, i.e. $K < 128$. Largest speedup of 1.8X happens to the group of $K = 32$. Performance gain comes from reducing un-

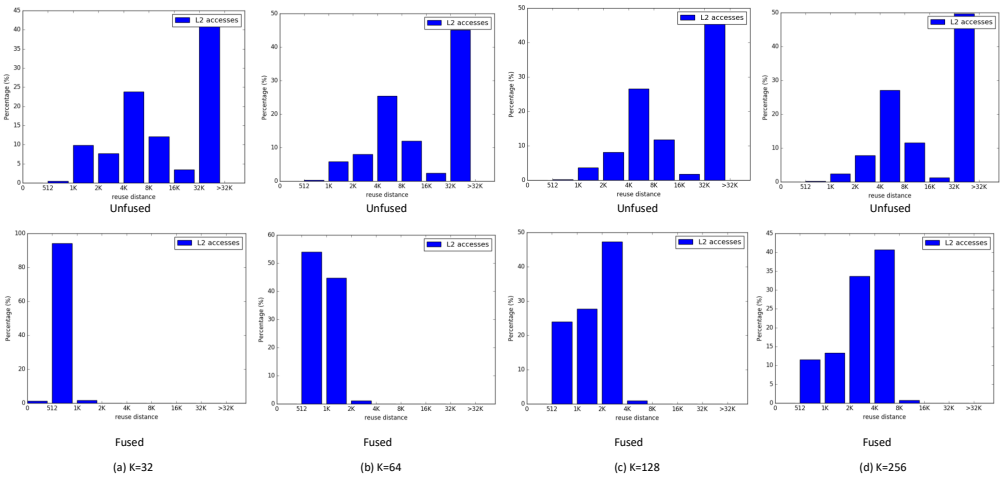


Figure 6.5: Reuse distance profile of CUDA-Unfused and Fused approach at shared L2. M=N=131072

necessary main memory accesses. As dimension K increases the performance degradation due to my inferior CUDA-C GEMM implementation outweighs the benefits of fused computation. The speedup against *CUDA-Unfused* is a projected speedup which suggests performance benefit of fusion when a GEMM as good as the one in cuBLAS is applied in *Fused*. *Fused* shows much better performance than *CUDA-Unfused* in all problem sizes. Compared to the *CUDA-Unfused* implementation, *Fused* gains a maximum 3.7X performance speedup when $K = 32$ and around 1.5X speedup when $K = 256$. This demonstrates the benefits of fusing over an unfused implementation. It is noticed that in lower dimension scenarios, performance benefit of fusion becomes more obvious as the number of points (M, N value) increases. Figure 6.5 compares the

Table 6.2: FLOP Efficiency

	cuBLAS-Unfused	*Fused*
K=32		
M=1024	19.92%	33.14%
M=131072	29.30%	50.86%
M=524288	29.02%	51.05%
K=64		
M=1024	31.15%	41.86%
M=131072	45.22%	57.01%
M=524288	36.83%	56.26%
K=128		
M=1024	44.32%	49.08%
M=131072	62.15%	60.03%
M=524288	61.76%	50.29%
K=256		
M=1024	58.42%	53.75%
M=131072	74.02%	62.9%
M=524288	74.15%	62.05%

reuse distance distribution seen at GPU shared L2 before and after fusion algorithm is applied. The figure shows that the fusion algorithm greatly reduces the data reuse distance. More than 50% of data in the unfused algorithm have data reuse longer than 32K, which accounts for 2MB cache. Fusion improves data locality by maximize data reuse and reduce the data reuse distance of all the L2 access to be less than 8K, which is equivalent of 512KB cache.

Table 6.2 demonstrates the ratio of achieved operations to peak single-precision floating-point operations, which is essentially flop efficiency. Since NVIDIA profiler reports efficiency value on the granularity of kernel launched, the efficiency of *cuBLAS-Unfused* kernel summation is a weighted sum of the

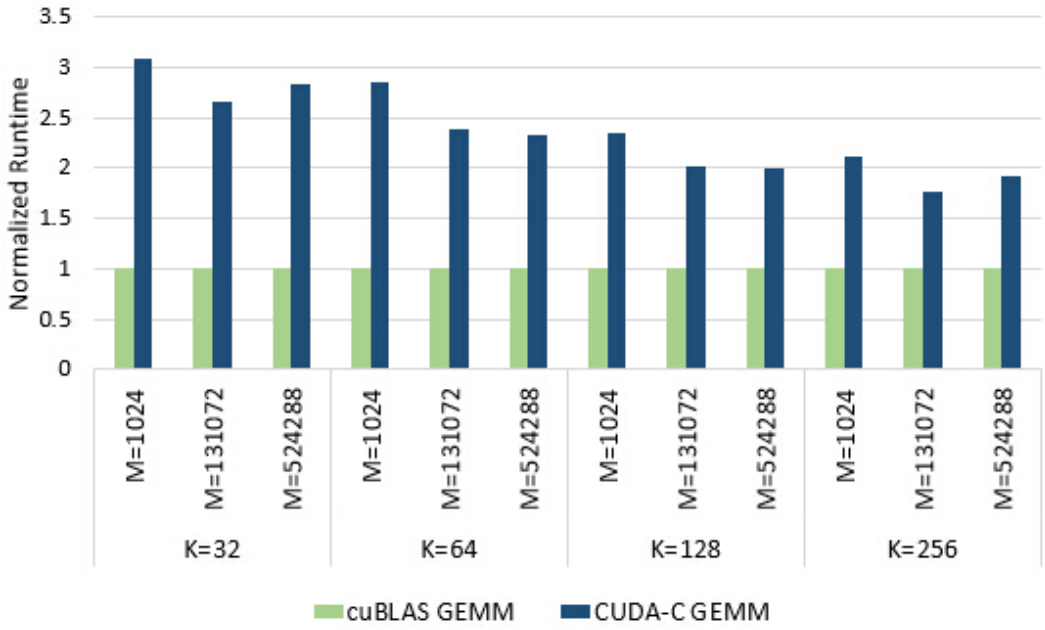


Figure 6.6: Execution time comparison of different GEMM implementations

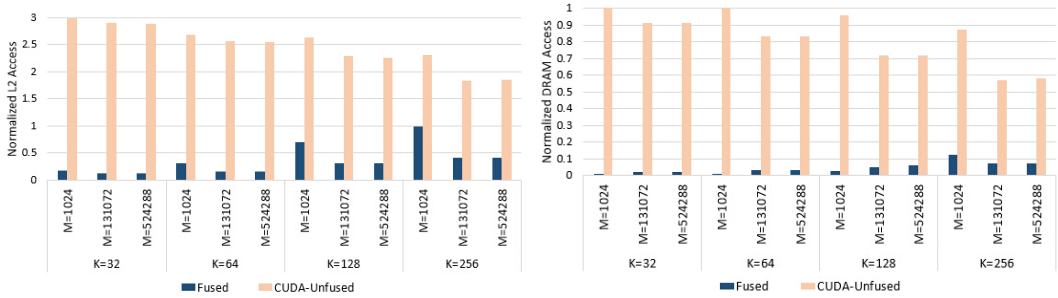
SGEMM kernel and the summation kernel based on their total cycle count. Higher FLOP efficiency indicates better performance. When the efficiency of fused kernel summation is lower than that of cuBLAS approach, the speedup over cuBLAS drops below 1X.

Since GEMM dominates the performance of kernel summation, a comparison between the cuBLAS GEMM and my CUDA-C GEMM would be helpful to better understand the overall performance. Figure 6.6 presents the normalized run time of the two GEMM implementation. As expected, the CUDA-C GEMM is two times slower than the cuBLAS GEMM. One of the main reasons of performance deterioration is the coarse-grained control of the CUDA-C language on hardware compared with assembly instructions. For

example, it is infeasible to avoid register file bank conflict when coding in the CUDA-C programming language and the *__syncthreads()* function is the primary synchronization method between threads, which is more expensive than the low level synchronization instructions available in the Maxwell assembly. Another reason that leads to inferior performance is that the part of storing results back to main memory is not optimized since it is unnecessary in kernel fusion. Apart from optimizing memory access ordering, rearranging the data location in shared memory to avoid bank conflicts, and applying float4 type of load/store instructions as many as possible, there are still some unknown optimization schemes in the cuBLAS library that contributes better performance.

6.4.1 Influence on data movement

Analyzing the performance of *Fused* kernel summation involves discussing the trade-offs between its lower GEMM performance and reduction in the number of memory accesses. The primary effect of the proposed code optimizations is the reduction in memory transactions. Figure 6.7 compares the number of L2 and DRAM transactions in both *Fused* and *CUDA-Unfused* normalized with respect to *cuBLAS-Unfused*. In Figure 6.7a, the number of L2 transactions in the *Fused* approach is less than 50% of the *cuBLAS-Unfused* approach in most cases, except for two configurations "M=N=1024, K=128" and "M=N=1024, K=256". In higher K-dimensional scenarios CUDA-C version of SGEMM has more L2 transactions compared with the SGEMM in



(a) L2 Transaction (b) DRAM Transaction

Figure 6.7: L2, DRAM transaction number normalized to *cuBLAS-Unfused*.

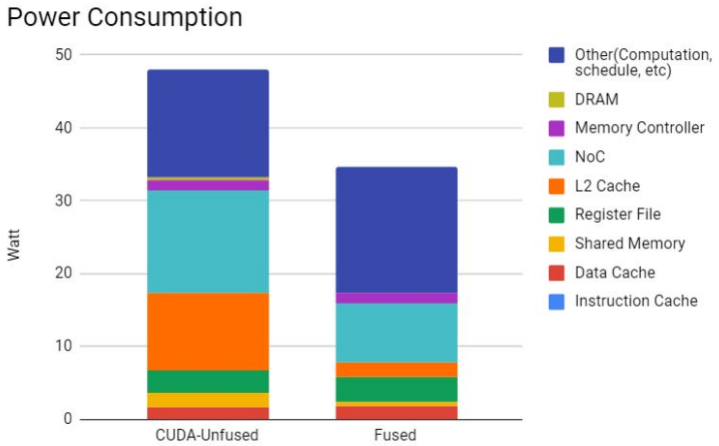


Figure 6.8: Power comparison between CUDA-Unfused and Fused approach. M=N=1024,K=256

cuBLAS library. In configurations where product of MN is small and K value is large, the benefit of saving L2 transactions through kernel fusion is offset by additional L2 transactions in SGEMM. As shown in Figure 6.7b, the number of DRAM transactions in *Fused* is less than 10% of *cuBLAS-Unfused* in all problem sizes.

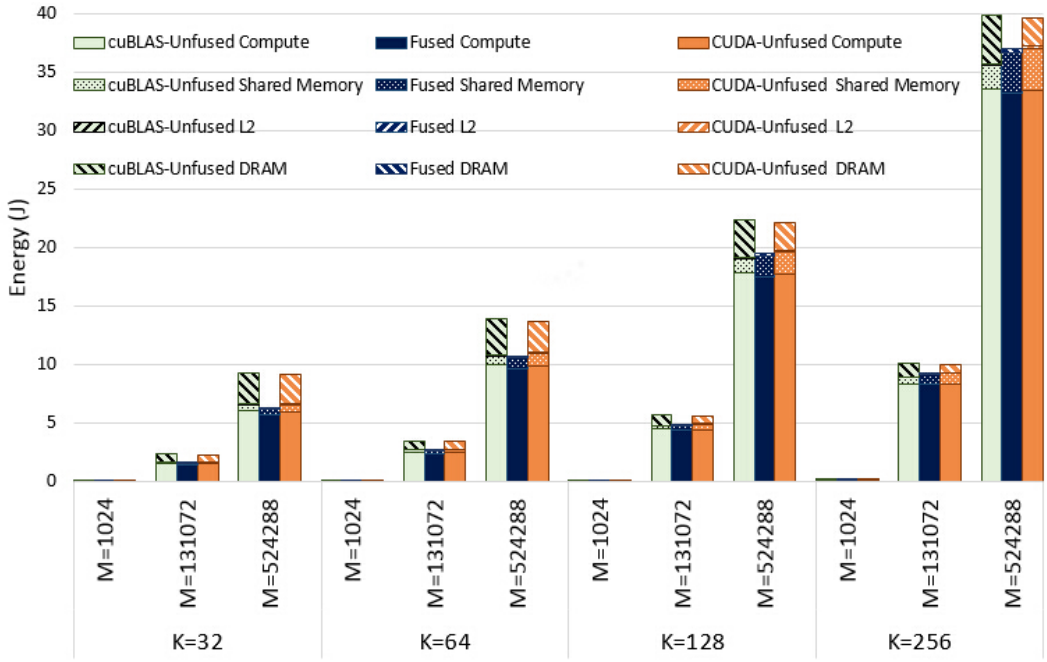


Figure 6.9: Energy consumption breakdown into Compute, Shared memory, L2, and DRAM

Reduction in L2 and DRAM accesses not only saves L2 and DRAM power and energy, but also power to move data around. Figure 6.8 illustrates total power consumption of *Fused* and *CUDA-Unfused* with detailed breakdown of various micro-architectural components. Fused kernel summation consumes 28% less power than unfused version. L2 cache and network on chip (NoC) are the two major power saving contributors.

6.4.2 Energy

In addition to performance benefit, fused kernel summation brings considerable energy savings thanks to reduction in main memory accesses. Ta-

Table 6.3: Energy Savings of *Fused* compared to *cuBLAS-Unfused*

	M=1024	M=131072	M=524288
K=32	31.3%	32.5%	32.5%
K=64	18.7%	23.6%	23.4%
K=128	10.2%	14.8%	13.1%
K=256	3.5%	8.5%	7.2%

ble 6.3 summarizes energy savings of the *Fused* approach compared to the *cuBLAS-Unfused*. Within in a group of same dimension K, there is a trend of saving more energy when the value of dimension M increases, which is because the number of redundant memory reads and writes are $O(MN)$ in kernel summation problem. The amount of energy savings obtained from fusion is greatly affected by the K value. Up to 33% of *cuBLAS-Unfused* energy is saved when $K = 32$, and energy savings decreases as value K increases, around 8% is saved when $K = 256$.

Figure 6.9 compares energy consumption of three different solutions, and illustrates energy breakdown into computation, shared memory, L2, and DRAM parts. Compared to the DRAM access energy in the *cuBLAS-Unfused* approach, the *Fused* approach saves more than 80% which amounts to 8% to 24% of total energy. The largest energy saving which is up to 33% happens to the group of $K = 32$. Out of 33%, DRAM access reduction contributes 26% , and the remaining 7% comes from reduction in the number of executed instructions. This is consistent with the performance speedup. When the *Fused* approach performance is better than the *cuBLAS-Unfused* approach, there is additional energy savings. In high dimension scenarios, the energy benefit

from fusion is less. One reason is because DRAM access savings will balance extra energy consumption from more shared memory accesses. Another reason is because more than 80% of energy is spent on floating point computing operations such as fused multiply add.

6.5 Summary

This chapter presents a fused approach of implementing kernel summation on the state of the art GPU. Various software optimizations to improve performance and energy efficiency are implemented into the kernel summation code. Fusing series of steps in the kernel summation leads to improvement in locality and reduction of memory accesses. In addition to fusion, steps of kernel summation are optimized to increase locality by adjusting the blocking and panel sizes, and by tailoring the working set to fit in the fast on-chip memory. Fusion is seen to improve overall performance of kernel summation up to 1.8X. This chapter shows that my approach achieves higher performance compared with the approach using the cuBLAS library in lower dimensions. Performance loss in high dimensions is due to my less efficient SGEMM. If an SGEMM as good as cuBLAS is applied, fused implementation is able to achieve up to 3.7X performance improvement. From the energy perspective, fused kernel summation shows 3% to 33% of total energy saving across various experimented dimensions. This is because eliminating redundant memory accesses via fusion results in less memory access energy. The fused approach always brings energy saving benefits and demonstrate optimizations at the

CUDA-C level while further improvements can possibly be obtained by optimizing the code at assembly level. Steps similar to those implemented in this chapter can be applied to other algorithms.

Chapter 7

Conclusion and Future Work

High memory subsystem performance and low energy consumption is the key to build energy efficient computing systems. Although there has been prior art on cache optimization, they still suffer from several challenges. First, the working set size of emerging applications has been continuing to grow and exceeding on-chip SRAM capacity, meanwhile the SRAM capacity per tenant is reaching its limit due to the power and area constraints. The increasing cache resource contention leads to high cache miss rate and poor memory subsystem performance. Second, as the cache hierarchy grows deeper, the energy cost of the large amount of data movement between cache layers has become negligible. Computation energy is almost free compared with data movement. The energy spent on memory subsystem, especially on data movement, has become the major source of system energy consumption. Third, with the trend of moving towards exclusive caches in CPU cache hierarchy, prior art proposed with inclusive caches in mind can not be directly applied without adding huge storage and power overhead. Finally, as GPU emerges as popular platform for high performance computing, proposed schemes in the filed of managing CPU data placement can not be easily applied due to GPU's extreme low per-thread SRAM capacity.

This dissertation focuses on improving energy efficiency of modern memory subsystem by proposing data movement sensitive data placement schemes. The following section summarizes the key contributions made in this dissertation.

7.1 Summary

The cache hierarchy is a highly contended resource in current multi-core, multi-tenant systems. With large data sets of modern applications and increasing core counts, the on-chip SRAM capacity per core is getting smaller. Installing data blocks to appropriate cache level and bypass layers in between contributes to energy saving, while better utilization of the free space in upper low-latency levels of the hierarchy together with quick correction of prediction errors is the key to performance improvement. In this dissertation, I present a FILtered Multilevel (FILM) caching policy, which is a PC guided mechanism to filter cacheline insertions based on data reuse with minimal cache overhead for meta-data transmission and storage. The bloom filter helps to overcome the challenges associated with capturing PC-based information in exclusive caches. When there is free space in the bypassed cache layer, FILM overrides the initial prediction and allows cache block insertion into the cache level achieving more low latency hits. FILM also incorporates an explicit mechanism for handling prefetch, which allows it to train differently for data from demand requests versus prefetch requests. As the locality behavior of memory instructions changes during application execution, FILM incorporates quick

detection and correction of stale/incorrect bypass decisions helping to achieve additional performance benefits.

Next, this dissertation investigates fine-grained data placement schemes for GPUs. The GPU has emerged as a powerful computing device for high-performance computing applications beyond graphics processing. While GPU provides incredible speedups for embarrassingly data parallel applications, it often requires extensive algorithm optimization effort for applications with high communication demands to achieve similar performance improvements as on other programs. Although many basic buildings blocks exist in CUDA library helping port applications from CPU to GPU, often a much more powerful solution is possible by tailoring these basic blocks to the specific application. Kernel summation is a widely used computational kernel that involves matrix-matrix multiplication (GEMM) and matrix-vector multiplication (GEMV) computational primitives. State of the art GPU solutions apply cuBLAS library but cannot exploit much of the data locality because intermediate results are written back to main memory in between key operations. This dissertation presents an optimized implementation that yields better performance and high energy efficiency. Data access sequence and placement is manually optimized based on the microarchitecture of GPU memory hierarchy. All steps of kernel summation are fused into the matrix multiplication code structure based on the temporal locality of data access pattern. The proposed approach achieves the best performance and least energy consumption via eliminating redundant data movement and main memory accesses.

7.2 Future Work

While this dissertation makes significant contributions to improve energy efficiency of memory subsystem via managing data placement, there are still opportunities for future work. This section lists possible future work.

Integrated CPU-GPU heterogeneous system is becoming a popular computing platform. Due to dramatic difference between the characteristics of CPU and GPU memory subsystems, data placement schemes designed for either CPU-only system or GPU-only system becomes inefficient for the CPU-GPU integrated system. As GPU generates much more larger data requests than CPU, any resources shared by CPU and GPU (e.g., LLC capacity, interconnect bandwidth, etc) will be dominated by GPU utilization, which leads to significant performance degradation in the usually latency-sensitive CPU workloads. A heterogeneous architecture aware data placement scheme is required to protect the cache performance of CPU from GPU pollution, meanwhile allocating enough resource to maintain the performance of bandwidth sensitive GPU applications.

The future heterogeneous aware data placement scheme should differentiate the cost of moving data within CPU or GPU cache hierarchy and the cost of moving data between CPU and GPU, where the latter cost is usually much higher than the former one. Besides, as CPU is much more latency sensitive than GPU and GPU is able to hide latency by sending large amount of data requests in consecutive cycles, the future scheme should protect data blocks required by CPU compute units from being replaced by GPU-required

data blocks. Future schemes may benefit from marking each cache block with its "owner" to be CPU or GPU.

Bibliography

[1] Cassandra. wiki.apache.org/cassandra/FrontPage.

[2] Faban Harness and Benchmark Framework. http://java.net/projects/faban/.

[3] Fermi compute architecture whitepaper. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[4] Maxas. https://github.com/NervanaSystems/maxas.

[5] MongoDB. mongodb.org.

[6] Profiler user's guide. http://docs.NVIDIA.com/cuda/profiler-users-guide/#axzz3oNg3mHRn.

[7] Tuning CUDA applications for maxwell. http://docs.nvidia.com/cuda/maxwell-tuning-guide/#axzz3op9EeX3M.

[8] VoltDB. http://voltdb.com.

[9] J. Ahn, S. Yoo, and K. Choi. Dasca: Dead write prediction assisted stt-ram cache architecture. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 25–36, Feb 2014.

[10] M. A. Z. Alves, , E. Ebrahimi, V. T. Narasiman, C. Villavieja, P. O. A. Navaux, and Y. N. Patt. Energy savings via dead sub-block prediction. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, pages 51–58, Oct 2012.

[11] Yuichiro Anzai. *Pattern Recognition & Machine Learning*. 2012.

[12] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayiran, Gabriel H Loh, Chita R Das, Mahmut T Kandemir, and Onur Mutlu. Exploiting inter-warp heterogeneity to improve GPGPU performance. In *24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 25–38, 2015.

[13] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[14] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002*, pages 73–78, May 2002.

[15] Nathan Beckmann and Daniel Sanchez. Modeling cache performance beyond lru. In *IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236, 2016.

[16] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. In *Journal of Computational Physics*, volume 231, pages 2825–2839, April 2012.

[17] Burton H. Bloom. Space/Time trade-offs in hash coding with allowable errors. In *Communication of ACM*, volume 13, pages 422–426, July 1970.

[18] Anand Chandrasekher. Meet qualcomm centriq 2400, the world's first 10-nanometer server processor. `https://www.qualcomm.com/news/onq/2016/12/07/meet-qualcomm-centriq-2400-worlds-first-10-nanometer-server-processor`.

[19] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 293–304, 2012.

[20] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. In *IEEE Transactions on Computers*, volume 44, pages 609–623, May 1995.

[21] Xi E. Chen and Tor M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and mshrs. In *41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 59–70, 2008.

[22] Hongwei Cheng, Leslie Greengard, and Vladimir Rokhlin. A fast adaptive multipole algorithm in three dimensions. In *Journal of Computational Physics*, volume 155, pages 468–498, 1999.

[23] D Yu Chenhan, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k nearest-neighbor kernel on x86 architectures. 2015.

[24] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. In *SIGOPS Oper. Syst. Rev.*, volume 43, pages 5–10, January 2010.

[25] Jamison D Collins and Dean M Tullsen. Hardware identification of cache conflict misses. In *32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 126–135, 1999.

[26] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. In *IEEE Micro*, volume 30, pages 16–29, March 2010.

[27] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

[28] CRC2-2017. THE 2ND CACHE REPLACEMENT CHAMPIONSHIP. `http://crc2.ece.tamu.edu/c`, 2017.

[29] B. Dally. Power, programmability, and granularity: The challenges of exascale computing. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 878–878, May 2011.

[30] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh ewald: An n log (n) method for ewald sums in large systems. In *The Journal of chemical physics*, volume 98, pages 10089–10092, 1993.

[31] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. Improving cache management policies using dynamic reuse distances. In *45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 389–400, 2012.

[32] Yoav Etsion and Dror G Feitelson. Exploiting core working sets to filter the l1 cache with random sampling. In *IEEE Transactions on Computers*, volume 61, pages 1535–1550, 2012.

[33] P. Faldu and B. Grot. Leeway: Addressing variability in dead-block prediction for last-level caches. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 180–193, Sept 2017.

[34] Priyank Faldu and Boris Grot. Leeway: Addressing variability in dead-block prediction for last-level caches. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 180–193, 2017.

[35] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *SIGPLAN Not.*, volume 47, pages 37–48, March 2012.

[36] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *25th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 25, pages 102–110, 1992.

[37] Adi Fuchs, Shie Mannor, Uri Weiser, and Yoav Etsion. Loop-aware memory prefetching using code block working sets. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 533–544, 2014.

[38] Wanling Gao, Yuqing Zhu, Zhen Jia, Chunjie Luo, Lei Wang, Zhiguo Li, Jianfeng Zhan, Yong Qi, Yongqiang He, Shimin Gong, Xiaona Li, Shujie Zhang, and Bizhu Qiu. Bigdatabench: a big data benchmark suite from web search engines. volume abs/1307.0320, 2013.

[39] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *ACM/IEEE 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 81–92, June 2011.

[40] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings 1999 International Symposium on Low Power Electronics and Design*, pages 70–75, Aug 1999.

[41] Alexander G Gray and Andrew W Moore. N-body'problems in statistical learning. In *NIPS*, volume 4, pages 521–527, 2000.

[42] John L. Henning. Spec cpu2006 benchmark descriptions. In *SIGARCH Comput. Archit. News*, volume 34, pages 1–17, September 2006.

[43] Johannes Hofmann, Georg Hager, Gerhard Wellein, and Dietmar Fey. An analysis of core- and chip-level architectural features in four generations of intel server processors. 02 2017.

[44] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. In *The Annals of Statistics*, pages 1171–1220, 2008.

[45] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. In *IEEE Micro*, volume 27, pages 63–72, 2007.

[46] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *SIGARCH Comput. Archit. News*, volume 30, pages 209–220, May 2002.

[47] MKL Intel. Intel math kernel library, 2007.

[48] Ciji Isen and Lizy John. Eskimo-energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem. In *42nd Annual IEEE/ACM IEEE/ACM International Symposium on Microarchitecture*, pages 337–346, 2009.

[49] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd International ACM Conference on International Conference on Supercomputing*, ICS '09, pages 499–500, 2009.

[50] A. Jain and C. Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89, June 2016.

[51] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 343–353, Feb 2015.

[52] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *SIGARCH Comput. Archit. News*, volume 38, pages 60–71, June 2010.

[53] Jonas Jalminger and P Stenstrom. A novel approach to cache block

reuse predictions. In *International Conference on Parallel Processing*, pages 294–302, 2003.

[54] Natalie D Enright Jerger, Eric L Hill, and Mikko H Lipasti. Friendly fire: understanding the effects of multiprocessor prefetches. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–188, 2006.

[55] W. Jia, K. A. Shaw, and M. Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–283, Feb 2014.

[56] Daniel A Jiménez. Dead block replacement and bypass with a sampling predictor. In *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[57] Daniel A Jiménez. Insertion and promotion for tree-based pseudolru last-level caches. In *46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 284–296, 2013.

[58] Daniel A. Jiménez and Elvira Teran. Multiperspective reuse prediction. In *50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 436–448, 2017.

[59] A. L. Narasimha Reddy Jinchun Kim, Paul V. Gratz. Lookahead

prefetching with signature path. http://comparch-conf.gatech.edu/dpc2/, 2015. [The 2nd Data Prefetching Championship (DPC2)].

[60] Ajay Joshi, Aashish Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. In *IEEE Transactions on Computers*, volume 55, pages 769–782, June 2006.

[61] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ACM/IEEE 17th International Symposium on Computer Architecture (ISCA)*, pages 364–373, May 1990.

[62] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ACM/IEEE 17th International Symposium on Computer Architecture (ISCA)*, pages 364–373, May 1990.

[63] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps,

J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2017.

[64] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ACM/IEEE 17th International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990.

[65] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. In *IEEE Micro*, volume 31, pages 7–17, Sep. 2011.

[66] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 56–65, Sep. 2013.

[67] Samira Manabi Khan, Yingying Tian, and Daniel A Jimenez. Sampling dead block prediction for last-level caches. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186, 2010.

[68] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and

bypassing algorithms. In *IEEE Transactions on Computers*, volume 57, pages 433–447, 2008.

[69] A. Khawaja, J. Wang, A. Gerstlauer, L. K. John, D. Malhotra, and G. Biros. Performance analysis of HPC applications with irregular tree data structures. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 418–425, Dec 2014.

[70] J. Kin, Munish Gupta, and W. H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 184–193, Dec 1997.

[71] Peter Kogge and John Shalf. Exascale computing trends: Adjusting to the "new normal" for computer architecture. In *Computing in Science & Engineering*, volume 15, pages 16–26, 11 2013.

[72] Partha Kundu, Murali Annavaram, Trung Diep, and John Shen. A case for shared instruction cache on chip multiprocessors running oltp. In *SIGARCH Comput. Archit. News*, volume 32, pages 11–18, September 2003.

[73] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. In *Communications of the ACM*, volume 55, pages 101–109, 2012.

[74] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn’t, and why. In *ACM Trans. Archit. Code Optim.*, volume 9, pages 2:1–2:29, March 2012.

[75] Junghoon Lee, Taehoon Kim, and Jaehyuk Huh. Dynamic prefetcher reconfiguration for diverse memory architectures. In *IEEE 34th International Conference on Computer Design (ICCD)*, pages 125–132, 2016.

[76] Shin-Ying Lee and Carole-Jean Wu. Ctrl-C: Instruction-aware control loop based adaptive cache bypassing for GPUs. In *IEEE 34th International Conference on Computer Design (ICCD)*, pages 133–140, 2016.

[77] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W Keller. Energy management for commercial servers. In *Computer*, volume 36, pages 39–48, 2003.

[78] Robert W Leland, Richard Murphy, Bruce A Hendrickson, Katherine Yelick, John Johnson, and Jonathan Berry. Large-scale data analytics and its relationship to simulation. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[79] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling energy optimizations in GPGPUs. In *SIGARCH Comput. Archit. News*, volume 41, pages 487–498, June 2013.

[80] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for GPUs. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[81] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th International ACM Conference on International Conference on Supercomputing*, pages 67–77, 2015.

[82] Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, and Xu Cheng. Optimal bypass monitor for high performance last-level caches. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 315–324, 2012.

[83] Mengjie Li, Matheus Ogleari, and Jishen Zhao. Logging in persistent memory: to cache, or not to cache? In *Proceedings of the International Symposium on Memory Systems*, pages 177–179, 2017.

[84] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, Dec 2009.

[85] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[86] Jieun Lim, Nagesh B Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. Power modeling for GPU architectures using McPAT. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, volume 19, page 26, 2014.

[87] J. Lin, Y. Chen, W. Li, A. Jaleel, and Z. Tang. Understanding the memory behavior of emerging multi-core workloads. In *2009 Eighth International Symposium on Parallel and Distributed Computing*, pages 153–160, June 2009.

[88] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, 2008.

[89] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, Al Geist, Rud Haring, Jeffrey Hittinger, Adolfy Hoisie, Dean Micron Klein, Peter Kogge, Richard Lethin, Vivek Sarkar, Robert Schreiber, John Shalf, Thomas Sterling, Rick Stevens, Jon Bashor, Ron

Brightwell, Paul Coteus, Erik Debenedictus, Jon Hiller, K. H. Kim, Harper Langston, Richard Micron Murphy, Clayton Webster, Stefan Wild, Gary Grider, Rob Ross, Sven Leyffer, and James Laros III. Doe advanced scientific computing advisory subcommittee (ascac) report: Top ten exascale research challenges. 2 2014.

[90] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *SIGPLAN Not.*, volume 40, pages 190–200, June 2005.

[91] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. In *Computer*, volume 35, pages 50–58, February 2002.

[92] Krishna T Malladi, Benjamin C Lee, Frank A Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. Towards energy-proportional datacenter memory with mobile DRAM. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 37–48, 2012.

[93] William B March and George Biros. Far-field compression for fast kernel summation methods in high dimensions. In *Applied and Computational Harmonic Analysis*, 2015.

[94] William B March, Bo Xiao, Chenhan D Yu, and George Biros. An algebraic parallel treecode in arbitrary dimensions. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 571–580, 2015.

[95] Ian Masliah, Ahmad Abdelfattah, A. Haidar, S. Tomov, Marc Baboulin, J. Falcou, and J. Dongarra. High-performance matrix-matrix multiplications of very small matrices. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, pages 659–671, 2016.

[96] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. In *IBM Syst. J.*, volume 9, pages 78–117, June 1970.

[97] Pierre Michaud. A best-offset prefetcher. `http://comparch-conf.gatech.edu/dpc2/`, 2015. [The 2nd Data Prefetching Championship (DPC2)].

[98] Sebastian Mika, Bernhard Schölkopf, Alexander J Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel pca and de-noising in feature spaces. In *NIPS*, volume 4, page 7, 1998.

[99] Sparsh Mittal. A survey of cache bypassing techniques. In *Journal of Low Power Electronics and Applications*, volume 6, page 5, 2016.

[100] Naohito Nakasato. A fast gemm implementation on the cypress GPU. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 50–55, 2011.

[101] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. In *International Journal of High Performance Computing Applications*, volume 24, pages 511–515, 2010.

[102] CUDA Nvidia. CuBLAS library. In *NVIDIA Corporation, Santa Clara, California*, volume 15, page 27, 2008.

[103] T. Palit, , and M. Ferdman. Demystifying cloud benchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, April 2016.

[104] R. Panda and L. K. John. Data analytics workloads: Characterization and similarity analysis. In *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pages 1–9, Dec 2014.

[105] R. Panda, X. Zheng, Jiajun Wang, A. Gerstlauer, and L. K. John. Statistical pattern based modeling of GPU memory access streams. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.

[106] Reena Panda and Lizy Zheng, Xinnian John. Accurate address streams for llc and beyond (SLAB): A methodology to enable system exploration. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 87–96, 2017.

[107] Dhinakaran Pandiyan. *Data Movement Energy Characterization of Emerging Smartphone Workloads for Mobile Platforms*. 2014.

[108] Vassilis Papaefstathiou, Manolis GH Katevenis, Dimitrios S Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, pages 325–334, 2013.

[109] J. J. K. Park, Y. Park, and S. Mahlke. A bypass first policy for energy-efficient last level caches. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 63–70, July 2016.

[110] Bhargavraj Patel, Nikos Hardavellas, and Gokhan Memik. Scp: Synergistic cache compression and prefetching. In *IEEE 33rd International Conference on Computer Design (ICCD)*, pages 164–171, 2015.

[111] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 10–20, March 2005.

[112] P Prinz, T Crawford, JL Hennessy, and DA Patterson. Computer architecture: A quantitative approach. 2018.

[113] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. f. Chuang, R. L. Scott, A. Jaleel, S. L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 626–637, Feb 2014.

[114] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 626–637, 2014.

[115] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical sram-tags with a simple and practical design. In *45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246, Dec 2012.

[116] Subramanian Ramaswamy and Sudhakar Yalamanchili. Customized placement for high performance embedded processor caches. In *Proceedings of the 20th International Conference on Architecture of Computing Systems*, ARCS'07, pages 69–82, 2007.

[117] A. Rizk, M. Zink, and R. Sitaraman. Model-based design and anal-

ysis of cache hierarchies. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9, June 2017.

[118] Arun F. Rodrigues, Gwendolyn Renae Voskuilen, and Simon David Hammond. *Multi-Level Memory: What You Add Is More Important Than What You Take Out.* Sep 2016.

[119] Suleyman Sair, Timothy Sherwood, and Brad Calder. A decoupled predictor-directed stream prefetching architecture. In *IEEE Transactions on Computers*, volume 52, pages 260–276, 2003.

[120] Ivo F Sbalzarini, Jens H Walther, Michael Bergdorf, Simone Elke Hieber, Evangelos M Kotsalis, and Petros Koumoutsakos. Ppm–a highly efficient parallel particle–mesh library for the simulation of continuum systems. In *Journal of Computational Physics*, volume 215, pages 566–588, 2006.

[121] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond.* 2002.

[122] A. Sembrant, E. Hagersten, and D. Black-Schaffer. A split cache hierarchy for enabling data-oriented optimizations. In *IEEE 23rd International Symposium on High Performance Computer Architecture (HPCA)*, pages 133–144, Feb 2017.

[123] Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. Data placement across the cache hierarchy: Minimizing data movement with

reuse-aware placement. In *IEEE 34th International Conference on Computer Design (ICCD)*, pages 117–124, 2016.

[124] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *48th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–152, 2015.

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

[126] Jaewoong Sim, Jaekyu Lee, Moinuddin K. Qureshi, and Hyesoon Kim. Flexclusion: Balancing cache capacity and on-chip bandwidth via flexible exclusion. In *SIGARCH Comput. Archit. News*, volume 40, pages 321–332, June 2012.

[127] Alan Jay Smith. Cache memories. In *ACM Computing Surveys (CSUR)*, volume 14, pages 473–530, 1982.

[128] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, O Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[129] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos.

Spatial memory streaming. In *ACM/IEEE 33rd International Symposium on Computer Architecture (ISCA)*, pages 252–263, 2006.

[130] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. In *Neural processing letters*, volume 9, pages 293–300, 1999.

[131] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. In *Proceedings of the IEEE*, volume 105, pages 2295–2329, Dec 2017.

[132] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of dgemm on fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 35, 2011.

[133] Elvira Teran, Yingying Tian, Zhe Wang, and Daniel A Jiménez. Minimal disturbance placement and promotion. In *IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA)*, pages 201–211, 2016.

[134] Yingying Tian, Sooraj Puthoor, Joseph L Greathouse, Bradford M Beckmann, and Daniel A Jiménez. Adaptive GPU cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUS*, pages 25–35, 2015.

[135] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R Pleszkun. A modified approach to data cache management. In *28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–103, 1995.

[136] A. Jaleel M. Qureshi V. Young, C. Chou. Ship++: Enhancing signature-based hit predictor for improved cache performance. `https://crc2.ece.tamu.edu/?page_id=53`, 2017.

[137] J. Wang, X. Dong, and Y. Xie. Oap: An obstruction-aware cache management policy for stt-ram last-level caches. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 847–852, March 2013.

[138] J. Wang, A. Khawaja, G. Biros, A. Gerstlauer, and L. K. John. Optimizing GPGPU kernel summation for performance and energy efficiency. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 123–132, Aug 2016.

[139] J. Wang, R. Panda, and L. K. John. SelSMaP: A selective stride masking prefetching scheme. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 369–372, Nov 2017.

[140] Jiajun Wang, Reena Panda, and Lizy John. Prefetching for cloud workloads: An analysis based on address patterns. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–172, 2017.

[141] Jiajun Wang, Reena Panda, and Lizy K. John. SelSMaP: A selective stride masking prefetching scheme. In *ACM Transactions on. Architecture and Code Optimization (TACO)*, volume 15, pages 42:1–42:21, October 2018.

[142] Z. Wang, D. A. Jimnez, C. Xu, G. Sun, and Y. Xie. Adaptive placement and migration policy for an stt-ram-based hybrid cache. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, Feb 2014.

[143] Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A. Jiménez. Wade: Writeback-aware dynamic cache management for nvm-based main memory system. In *ACM Trans. Archit. Code Optim.*, volume 10, pages 51:1–51:21, December 2013.

[144] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Design, Automation and Test in Europe*, pages 600–605 Vol. 1, March 2005.

[145] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Making address-correlated prefetching practical. In *IEEE micro*, volume 30, 2010.

[146] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Making address-correlated prefetching practical. In *IEEE micro*, volume 30, 2010.

[147] Wikipedia-contributors. Epyc. https://en.wikipedia.org/wiki/Epyc, 2018.

[148] Wikipedia-contributors. List of intel xeon microprocessors. https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors#Skylake-based_Xeons, 2018.

[149] Theodore M Wong and John Wilkes. My cache or yours?: Making storage more exclusive. In *USENIX Annual Technical Conference, General Track*, pages 161–175, 2002.

[150] C. J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer. Ship: Signature-based hit predictor for high performance caching. In *44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441, Dec 2011.

[151] Youfeng Wu, R. Rakvic, Li-Ling Chen, Chyi-Chang Miao, G. Chrysos, and J. Fang. Compiler managed micro-cache bypassing for high performance epic processors. In *35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 134–145, 2002.

[152] Lingxiang Xiang, Tianzhou Chen, Qingsong Shi, and Wei Hu. Less reused filter: improving l2 cache performance via filtering less reused lines. In *Proceedings of the 23rd International ACM Conference on International Conference on Supercomputing*, pages 68–79, 2009.

[153] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for GPUs. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88, Feb 2015.

[154] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated static and dynamic cache bypassing for GPUs. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88, 2015.

[155] Chenhan D Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7, 2015.

[156] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive cache and concurrency allocation on GPGPUs. In *IEEE Computer Architecture Letters*, volume 14, pages 90–93, July 2015.

Vita

Jiajun Wang was born in Tianjin, China on 11 February 1991. She received the Bachelor of Science degree in Engineering from the Zhejiang University in May 2013. From August 2013, she started his Doctoral studies at The University of Texas at Austin. She has completed several internships. She interned with the performance analysis team of Centaur Technology twice during the summer of 2015 and 2016 in Austin, Texas. In 2017 summer, she worked as a research intern with Memory Team at arm in Austin, Texas. She also interned with the System-On-Chip Architecture team at Google in Mountain View, California in 2018.

Permanent address: jiajunw91@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.