The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

A STUDY OF PARALLELISM-LOCALITY TRADEOFFS ACROSS MEMORY HIERARCHY

A Dissertation in

Computer Science and Engineering

by

Praveen Yedlapalli

© 2015 Praveen Yedlapalli

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2015

The dissertation of Praveen Yedlapalli was read and approved[1] by the following:

Mahmut Taylan Kandemir
Professor of Computer Science and Engineering
Dissertation Adviser
Chair of Committee

Chita R. Das
Professor of Computer Science and Engineering

Padma Raghavan
Professor of Computer Science and Engineering

Dinghao Wu
Assistant Professor of Information Sciences and Technology

Lee Coraor
Associate Professor of Computer Science and Engineering
Chair of the Graduate Program

[1] Signatures on file in the Graduate School.

Abstract

As the number of cores on a chip increases, the memory bandwidth requirements become a scalability issue. Current CMPs incorporate multiple resources both on-chip and off-chip to handle these bandwidth requirements. There are multiple ways to organize these resources (caches and memory) with different parallelism and locality tradeoffs. In this dissertation, we first study parallelism vs. locality tradeoffs in each layer of the memory hierarchy, as well as the cross-layer interactions.

Using the observations from the characterization study we proposed a dynamic memory migration technique which optimizes both parallelism and locality metrics in the memory subsystem and thereby improve performance. Then we study the challenges faced by traditional cache prefetchers in modern CMPs and identify the major pitfalls in their use in these new systems. We show how memory prefetching can take advantage of the memory locality and prefetch opportunistically, leading to better efficiency than traditional cache prefetchers. We explore the emerging area of mobile computing and identify mobile memory bandwidth requirement as a major challenge faced in these systems. We propose a novel solution of breaking the application frames into smaller ones to exploit the memory locality and reduce the memory bandwidth requirements significantly in such systems.

Table of Contents

List of Tables vi

List of Figures vii

Acknowledgments ix

Chapter 1. Introduction 1
- 1.1 Memory Migration 3
- 1.2 Memory Prefetching 3
- 1.3 Mobile Memory 4
- 1.4 Programming Wall Challenge 6

Chapter 2. Background and Related Work 7
- 2.1 Background 7
  - 2.1.1 Mobile Platforms 9
- 2.2 Related Work 11

Chapter 3. Memory Migration 16
- 3.1 Parallelism vs. Locality 16
  - 3.1.1 Mapping Scenarios 16
  - 3.1.2 L2 Cache 16
  - 3.1.3 Memory Channel 18
  - 3.1.4 Memory Bank 20
  - 3.1.5 Summary of Findings 21
- 3.2 Dynamic Migration for Improving Memory Performance 22
  - 3.2.1 Migration Policy 22
  - 3.2.2 Migration Mechanisms 23
- 3.3 Experimental Evaluation 25
  - 3.3.1 Migration Policy 25
  - 3.3.2 Migration Mechanisms 26

Chapter 4. Memory Prefetching 28
- 4.1 Memory-Side Prefetching 29
  - 4.1.1 What to Prefetch? 29
  - 4.1.2 When to Prefetch? 30
  - 4.1.3 Where to Prefetch? 31
  - 4.1.4 Optimizations for Memory-Side Prefetching 32
- 4.2 Experimental Evaluation 33
  - 4.2.1 Setup 33
  - 4.2.2 Benchmarks 33
  - 4.2.3 Results and Analysis 35

Chapter 5. Mobile Memory 38
5.1 Problem Statement 38
5.2 Evaluation Platform 41
5.3 IP-to-IP Data Reuse 42
5.3.1 Data Reuse and Reuse Distance 42
5.3.2 Converting Data Reuse into Locality 43
5.4 Sub-Framing 44
5.4.1 Flow-Buffering 46
5.4.2 IP-IP Short-circuiting 47
5.4.3 Effects of Sub-framing Data with Flow-Buffering and IP-IP Short-circuiting 49
5.5 Implementation Details 49
5.5.1 Correctness 49
5.5.2 OS and Hardware Support 52
5.6 Evaluation 53

Chapter 6. Cooperative Parallelization 56
6.1 Parallelization Approach 56
6.1.1 Trees and Recursion 57
6.1.2 Linked Lists and Loops 59
6.2 Programmer Directives and Automation 61
6.2.1 Programmer Directives 61
6.2.2 Automation 63
6.3 Experimental Evaluation 64
6.3.1 Platform 64
6.3.2 Benchmarks 64
6.3.3 Results 65

Chapter 7. Future Work 67
7.1 Cooperative Prefetching 67
7.2 Mobile Memory Systems 67

Chapter 8. Concluding Remarks 68

Appendix. Publications 69
A.1 Significant Publications 69
A.2 Other Publications 69

Bibliography 71

List of Tables

2.1 Breakdown of on-chip and off-chip latencies for L2 misses in three different workloads without any prefetching. 8

3.1 Evaluated mapping scenarios. 17

4.1 Characteristics of different prefetch schemes. 31
4.2 Configuration of the evaluation platform. 33
4.3 Memory characteristics of SPEC2006 applications. 34

5.1 Platform configuration. 41
5.2 Expansions for IP abbreviations. 42
5.3 IP flows in our applications. 42

6.1 Hardware and Software configuration of the experimental evaluation platform. 64
6.2 Benchmark applications and their important properties. 65

List of Figures

2.1 Target SoC platform with a high-level view of different functional blocks in the system. 9
2.2 Overview of data flow in SoC architectures. 10

3.1 L2 parallelism under different mappings. (Higher is better) 18
3.2 L2 Locality under different mappings. (Lower is better) 18
3.3 Normalized IPC under different mappings. 19
3.4 Memory channel parallelism under different mappings. (Higher is better.) 19
3.5 Memory channel locality under different mappings. (Lower is better.) . 20
3.6 Normalized IPC under different mappings. 20
3.7 Normalized IPC under different memory mappings. 21
3.8 Percentage of data to be migrated already present in the L2 cache. . . . 24
3.9 Ring network connecting the memory controllers is used for data migration. 24
3.10 Performance improvements with different migration policies. 25
3.11 Improvements in memory parallelism obtained by our migration scheme. 26
3.12 Memory locality with our migration scheme. 26
3.13 Performance improvements with different migration mechanisms. 27
3.14 Memory latency with different migration mechanisms. 27

4.1 Line access pattern graphs of some SPEC CPU2K6 applications. 30
4.2 Percentage IPC improvement over no prefetching with different prefetchers. 35
4.3 Average on-chip and off-chip latencies for an LLC miss without prefetching and different prefetching schemes. 36

5.1 Bandwidth usage of Youtube and Skype over time. 39
5.2 Total data stalls and processing time in IPs during execution. 39
5.3 Percentage of frames completed in a subset of applications with varying memory bandwidths. 40
5.4 Trends showing increase of percentage of data stalls with each newer generation of IPs and DRAMs. 40
5.5 Percentage reduction in Cycles-Per-Frame in different flows with a perfect memory configuration. 40
5.6 Data access pattern of IPs in YouTube application. 43
5.7 Hit rates under various cache capacities. 44
5.8 Cycles Per Frame under various cache capacities. 45
5.9 Area and power-overhead with large shared caches. 45
5.10 IP-to-IP reuse distance variation with different sub-frame sizes. Note that the y-axis is in the log scale. 46
5.11 Delay breakdown of a memory request issued by IPs or cores. The numbers above the bar give the absolute cycles. 48
5.12 Hit rates with flow-buffering and IP-IP short-circuiting. 49

5.13 Pictorial representation showing the structure of five consecutive video frames. . . . 50
5.14 High level view of the SA that handles sub-frames. . . . 51
5.15 Percentage of Frames Completed (Higher the better). . . . 53
5.16 Reduction in Cycles Per Frame in a flow normalized to Baseline (Lower the better). . . . 54
5.17 Reduction in Number of Active Cycles of Accelerators (Lower the better). 55

6.1 A high level view of the proposed approach. . . . 56
6.2 High level code of a parallelized program using the proposed strategy. . 57
6.3 Subproblems in a tree-based application. . . . 58
6.4 A function from perimeter Rogers et al. (1995), one of our tree-based applications. . . . 59
6.5 A function from em3d Rogers et al. (1995), one of our linked list-based applications. . . . 60
6.6 Programming directives to express parallelism. Note that, *val* and *number* are optional fields. . . . 61
6.7 High level view of automation. . . . 63
6.8 Speedups obtained with our approach 66

Acknowledgments

I sincerely thank my advisor, Professor Mahmut Kandemir for his continued support through out my PhD. His guidance was very helpful in driving me in the correct direction. I learnt a lot of valuable lessons from him both in research and life. I admire his patience in reviewing my results and writing many times and providing valuable feedback. I appreciate the way he supported me when I wanted to change my thesis topic mid-way. I am extremely thankful to him for that.

I am thankful to my dissertation committee members Professor Chita R. Das, Professor Padma Raghavan and Professor Dinghao Wu for their valuable time and the helpful feedback. I am especially thankful to Dr. Chita Das and Dr. Anand Sivasubramaniam for their prolonged guidance in many of my projects. Interactions with them has always been immensely insightful and helpful to me.

I am very thankful to my collaborators Nachiappan Chidambaram and Emre Kultursay. Discussions with them are always interesting and gave me a new perspective on the problems. I am also thankful to my lab mates and collaborators Jagadish Kotra, Niranjan Soundararajan, Wei Ding, Adwait Jog, Diana Guttman, Jithendra Srinivas, Sai Prashant Muralidhara and Yuanrui Zhang.

I should thank VMware and Intel corporations for offering me summer internships that proved to be invaluable. Those were my first jobs in industry and I learnt a lot from them. It was a great experience interacting with people from different universities and that provided me a different perspective on academics and life.

I am also thankful to the organizations National Science Foundation (NSF), Department of Energy (DOE), Intel and Microsoft for the support they provided to our lab. Specifically, the NSF grants - #1302557, #0963839, #1205618, #1213052, #1320478, #1317560, #1302225, #1017882, #1152479, #1147388, #1139023, #0811687 and #0953246.

Finally, I want to thank my sister (Dr. Swathi Yadlapalli), my parents (Dr. Shailaja Yadlapalli and Dr. Prasad Yadlapalli) and my grand parents (Dr. Sarojini Yerneni and Dr. Venkateswara Rao Yerneni) for inspiring me to get a PhD and providing me motivation through out my time in PhD.

Chapter 1

Introduction

Chip multi-processors (CMPs) have become ubiquitous in all forms of computing starting from mobile platforms to data centers. They handle the power and instruction level parallelism (ILP) issues much better than the single-core processors. However, there are some important challenges to be addressed to use CMPs effectively. Traditionally, memory is slower compared to the processor. Putting more cores in a single processor makes the memory problem even worse. This problem is referred as the **Memory wall** challenge. In this dissertation, we look at the challenge from a memory parallelism vs locality perspective. We propose novel solutions to address this challenge for systems ranging from mobile computing domains to high performance platforms.

The memory wall continues to plague the scalability of high end computing systems. While this problem has persisted through the past decades of single core systems, the multicore era only exacerbates the increasing reliance on an uninterrupted supply of data (from the memory system) to keep the pipelines of different cores continuously busy. Current and next generation chip multiprocessors (CMPs) do offer the potential of better locality by accommodating deeper (2-3 levels) cache hierarchies on the same die as the computation units before requiring off-chip accesses. They also offer multiple (parallel) paths for getting to the memory hierarchy that is necessary to sustain the high throughput needs of multiple cores on the die. However, these two dimensions – *locality* and *parallelism* – need careful scrutiny and optimization for the successful scalability of next generation CMPs. There have been specific studies looking to optimize one or more artifacts of the memory hierarchy – the cache layers Kim et al. (2002); Cade and Qasem (2009); Huh et al. (2007); Mutlu et al. (2003), the memory controllers Jeong et al. (2012a); Sudan et al. (2010) and DRAM organization Kim et al. (2012); Udipi et al. (2010) or even the on-chip interconnect for accessing the hierarchy Das et al. (2009); Park et al. (2008). However, there has been little attempt at systematically studying these two dimensions of locality and parallelism, and the tradeoffs therein, across the layers of the memory hierarchy of a CMP. This dissertation presents the first such characterization, studying the locality vs. parallelism trade-offs that exist in each layer, as well as across layers. Note that, in this dissertation we use the term *locality* to represent the proximity of the cache bank from the processor.

Next generation CMPs are envisioned to have dozens, or even hundreds, of cores on the die. Each core typically has a local and private L1 cache (of the order of 16-32 KB). This L1 cache is fairly adequate to sustain the response time (access times of 1-2 cycles) and throughput (to handle memory requests which may be spaced a few instruction apart) needs of the associated core, even if it is of multiple issue, for the data that it holds. Consequently, the locality and parallelism of the L1 itself is not that much of a problem. However, the problems arise when the requests miss in the L1 cache. The last

level cache (LLC), whose capacity can run to about 1 MB (or more) for each bank, can take several cycles to service a request. Further, with this larger capacity, there is a much higher possibility of uneven-ness of usage/requirements across the different cores. A lot of recent work Kim et al. (2003, 2002) has thus looked at making the LLC shared[1], to allow better multiplexing the capacity across the nodes within this overall shared space. An L1 request miss could thus have differentiated behavior depending on whether it (i) hits in an LLC bank that is associated with the same requesting core (typically taking around 5-8 cycles), (ii) needs to get to another LLC bank on the die containing that data item after traversing an on-chip network (taking dozens of cycles), or (iii) misses in on-chip LLC (whether local or remote) and needs to go to a memory controller traversing the on-chip network as well as incurring off-chip DRAM latencies (taking hundreds of cycles). Owing to the high overheads, the locality and parallelism issues, in all these cases, can cause tremendous impact on the response times and throughput of the requests, with trade-offs between these two dimensions themselves.

Let us first examine these trade-offs at the LLC level. Having all of a core's L1 misses served at its local LLC bank is highly desirable from the locality viewpoint. Hence a private LLC is locality tuned, but may suffer from load imbalance issues (overutilization). If another core is less reliant on LLC, it may be better to have a shared LLC pool, and thus multiplex the needs of different cores. From a fully private LLC, we can go to an SNUCA-like organization Kim et al. (2003) where a core's data gets striped across the different LLC banks. The granularity of such striping would have a consequence on the locality and parallelism trade-offs. Finer the granularity, the better is the distribution of the load across all the LLCs on the die, and the better is the aggregate ability of the LLC banks to service multiple requests (possibly from different cores). We could go down to as fine a striping granularity as a cache block, which is essentially an S-NUCA LLC organization that has been extensively studied Kim et al. (2003, 2002). This parallelism and load balance comes at a cost of a loss in locality – a line may now need to be retrieved from any other LLC on-chip (not just the local one) requiring traversal through the on-chip network. The spatial locality is also only at the cache line granularity, since successive cache lines involve going to a different LLC banks. Between the private LLC and the cache-line interleaved shared LLC, we could also consider other striping granularities, such as a page, for a spectrum of locality vs. parallelism trade-offs.

Going down to the next level of a LLC miss, we can also explore locality vs. parallelism trade-offs at the memory controllers. We could opt to place all of a core's (application's) data on a single memory controller – preferably one that is closest to the LLC bank that receives a major portion of its L1 misses. Such a design may be preferential for locality since the distance (network hops) to get to the memory controller upon an LLC miss may be reduced. However, the same memory controller serves all of that application's requests, and this may be undesirable for a memory-intensive application which could benefit from multiple controllers (parallelism) serving its memory requests. Instead, we could again stripe the data of an application across the memory controllers,

[1]Without loss of generality, we consider the last level cache to be L2, and consider both private and shared options for it.

at page or even cache line levels for boosting parallelism, though this may hurt the locality behavior. Finally, once a request gets to a memory controller there are again locality vs. parallelism trade-offs when placing data across DRAM banks. Spreading it across independent banks allows concurrent transfers. Within a bank, locality can be exploited by transferring several cache lines at the same time into a row buffer, and subsequently reading them off without incurring additional overheads (for a open row policy).

To summarize, there are both cache (LLC) and memory level locality vs. parallelism trade-offs when designing a memory hierarchy for a CMP. These trade-offs are very much dependent on the characteristics of the workload running on the CMP – the memory intensity, the heterogeneity of this intensity across the cores, the spatial locality, the relative importance of latency versus throughput on the application, etc. Understanding these tradeoffs towards designing a scalable hierarchy for the next generation CMPs requires a thorough characterization using a spectrum of workloads. We are not aware of prior work that has systematically done such a characterization. With this primary motivation, this dissertation conducts a detailed characterization and analysis of the parallelism and locality trade-offs in the memory hierarchy of a network-on-chip (NOC) based CMP, using a spectrum of multiprogrammed and multithreaded workloads on a 32 core CMP. We analyze not just the trade-offs within each layer of the hierarchy (the LLC or main memory) but also the cross-layer interactions.

1.1 Memory Migration

Based on the characterization study, we next explore the possibility of a data migration scheme which can optimize both parallelism and locality in the memory subsystem. Our data migration policy exploits the parallelism-locality tradeoff in memory to improve the system performance. We build a dynamic runtime system which monitors the application's memory access patterns and determines the best address mapping strategies at each level in the memory hierarchy. To implement this policy, we also present novel migration mechanisms that can efficiently handle the data migration overheads.

1.2 Memory Prefetching

The growing scale of chip multiprocessors (CMPs), especially Network-On-Chip (NOC) based CMPs is magnifying the memory wall problem in several ways: (i) a larger number of hops has to be traversed in the on-chip network to get to the memory controllers before a request can even be issued to the memory (DRAM); (ii) the increased data demand from the cores causes higher contention for on-chip cache and network resources; (iii) higher contention at the memory controllers results in larger queuing delays; and (iv) the inter-mixing of requests across cores can lead to poor locality across pages in the row buffers of the DRAMs, thereby incurring higher latencies to switch the required rows before serving requests. All these detrimental factors accentuate the memory wall problem over and beyond the growing bandwidth disparity between the on-chip compute versus the off-chip memory.

One well-studied technique to address the memory wall problem is *prefetching*, both software Luk et al. (2002); Ortega et al. (2002); Wu (2002) and hardware Joseph and Grunwald (1997); Liu et al. (2011), wherein memory requests are (predicted and) issued ahead of their need so that the corresponding instructions find the requested data in an appropriate level of the cache when they execute. The prefetch accuracy, in terms of both *what* and *when*, is a key determinant to hiding the memory latency. In fact, studies Srinath et al. (2007); Zhuang and Lee (2003) have shown that aggressive and inaccurate prefetches can even degrade performance. Further, prefetches in the context of large CMPs have additional ramifications. They can increase memory bandwidth requirements Srinath et al. (2007) and network traffic (i.e., lead to higher contention) Chen and Baer (1994); Ebrahimi et al. (2009), and cause higher interference over cache lines Wu et al. (2011); Zhuang and Lee (2003), worsening the performance even further.

An earlier idea that has been explored in the context of uniprocessors is "memory-side" prefetching Hughes and Adve (2005); Hur and Lin (2006); Joseph and Grunwald (1997); Solihin et al. (2003); Yang and Lebeck (2000) where the memory system observes reference stream and pushes appropriate lines into the cache. This idea was primarily proposed to address the storage space and logic needed to maintain large on-chip predictor tables and/or to reduce the critical path of prefetch messages.

We propose a memory-side prefetcher that is integrated with the memory controllers of an on-chip network based CMP. It maintains a relatively small prefetch buffer of about 256KB per controller, into which our logic prefetches cache lines from DRAM row buffers based on various factors – predictability, bandwidth availability, row buffer conflict overheads, utility to different request streams, etc.

1.3 Mobile Memory

The propensity of tablets and mobile phones in this handheld era raises several interesting challenges. At the application end, these devices are being used for a number of very demanding scenarios (unlike the mobiles of a decade ago), requiring substantial computational resources for real-time interactivity, on both input and output sides, with the external world. On the hardware end, power and limited battery capacities mandate high degrees of energy efficiencies to perform these computational tasks. Meeting these computational needs with the continuing improvements in hardware capabilities is no longer just a matter of throwing high performance and plentiful cores or even accelerators at the problem. Instead, a careful examination and marriage of the hardware with the application and execution characteristics is warranted for extracting the maximum efficiencies. In other words, a co-design of software and hardware is necessary to design energy- and performance-optimal systems, which may not be possible just by optimizing the system in parts. With this philosophy, this dissertation focusses on an important class of applications run on these handheld devices (real-time frame-oriented video/graphics/audio), examines the data flow in these applications through the different computational kernels, identifies the inefficiencies when sustaining these flows in today's hardware solutions, which simply rely on main memory to exchange such data, and proposes alternate hardware enhancements to optimize such flows.

Real-time interactive applications, including interactive games, video streaming, camera capture, and audio playback, are amongst the most popular on today's tablets and mobiles apart from email and social networking. Such applications account for nearly 65% of the usage on today's handhelds mob (January-February 2013), stressing the importance of meeting the challenges imposed by such applications efficiently. This important class of applications has several key characteristics that are relevant to this study. First, these applications work with input (sensors, network, camera, etc.) and/or output (display, speaker, etc.) devices, mandating real-time responsiveness. Second, these applications deal with "frames" of data, with the requirement to process a frame within a stipulated time constraint. Third, the computation required for processing a frame can be quite demanding, with hardware accelerators[2] often deployed to leverage the specificity in computation for each frame and delivering high energy efficiency for the required computation. The frames are then pipelined through these accelerators one after another sequentially. Fourth, in many of these applications, the frames have to *flow* not just through one such computational stage (accelerator) but possibly though several such stages. For instance, consider a video capture application, where the camera IP may capture raw data, which is then encoded into an appropriate form by another IP, before being sent either to a flash storage or a display. Consequently, the frames have to flow through all these computational stages, and typically the memory system (the DRAM main memory) is employed to facilitate this flow. Finally, we may need to support several such flows at the same time. Even a single application may have several concurrent flows (the video part and audio part of the video capture application which have their own pipelines). Even otherwise, with multiprogramming increasingly prevalent in handhelds, there is a need to concurrently support individual application flows in such environments.

Apart from the computational needs for real-time execution, all the above observations stress the memory intensity of these applications. Frames of data coming from any external sensor/device is streamed in to memory, from which it is streamed out by a different IP, processed and put back in memory. Such an undertaking places heavy demands on the memory subsystem. When we have several concurrent flows, either within the same application or across applications in a multiprogrammed environment, all of these flows contend for the memory and stresses it even further. This contention can have several consequences: (i) without a steady stream of data to/from memory, the efficiencies from having specialized IPs with continuous dataflow can get lost with the IPs stalling for memory; (ii) such stalls with idle IPs can lead to energy wastage in the IPs themselves; and (iii) the high memory traffic can also contend with, and slow down, the memory accesses of the main cores in the system. While there has been a lot of work covering processing – whether it be CPU cores or specialized IPs and accelerators (e.g. Saleh et al. (2006)Zhu and Reddi (2013)Lee and Chang (2006)) – for these handheld environments, the topic of optimizing the data flows, while keeping the memory system in mind, has drawn little attention. Optimizing for memory system performance, and minimizing consequent queueing delays has itself received substantial interest in the past decade, but only in the area of high-end systems (e.g., Balasubramanian et al. (2009)

[2]We use the term accelerators and IPs interchangeably in this work.

Kim et al. (2010a) Kim et al. (2010b) Das et al. (2010)). This dissertation addresses this critical issue in the design of handhelds, where memory will play an increasingly important role in sustaining the data flow not just across CPU cores, but also between IPs, and with the peripheral input-output (display, sound, network and sensors) devices.

In today's handheld architectures, a *System Agent* (SA) Keltcher et al. (2003); Conway et al. (2010); Naveh et al. (2011); Yuffe et al. (2011) serves as the glue integrating all the compute (whether it be IPs or CPU cores) and storage components. It also serves as the conduit to the memory system. However, it does not clearly understand data flows, and simply acts as a slave initiating and serving memory requests regardless of which component requests it. As a result, the high frame rate requirements translate to several transactions in the memory queues, and the flow of these frames from one IP to another explicitly goes through these queues, i.e., the potential for data flow (or data *reuse*) across IPs is not really being exploited. Instead, in this dissertation we explore the idea of *virtually integrating accelerator pipelines* by "short-circuiting" many of the read/write requests, so that the traffic in the memory queues can be substantially reduced. Specifically, we explore the possibility of *shared buffers/caches* and *short-circuiting communication* between the IP cores based on requests already pending in the memory transaction queues.

1.4 Programming Wall Challenge

An industry wide transition from single-core to multicore processors IBM (2009); Intel (2009); AMD (2009) has brought the problem of program parallelization to forefront. Software developers can no longer rely on computer architects to speed up single threaded applications by increasing the clock frequency. Instead, developers are now obliged to build parallel programs to take full advantage of the underlying multicore architecture. However, building parallel programs manually is both challenging and error-prone. Another problem with the idea of manual parallelization is the existence of huge amount of legacy code which is mostly single threaded. The fact that it is very difficult to understand and parallelize programs developed by someone else is yet another reason why manual parallelization is very problematic on a large scale. Parallelizing compilers can offer some help through automatic parallelization of programs. Automatic parallelization has been a well studied problem Lamport (1974); Wolfe (1995); Banerjee (1988, 1994), which can be used as a solution to reduce the cost and time consumed in developing parallel programs.

We propose an automatic parallelization technique for pointer-based applications. It is a cooperative technique between the programmer, the compiler and the runtime system to identify and efficiently exercise parallelism. The programmer provides hints that indicate the parallelism in the code. The compiler understands the hints and generates the parallel code. And the runtime system monitors the program and efficiently executes the parallel code.

Chapter 2

Background and Related Work

2.1 Background

A typical commonality of applications working on dynamic data structures is that, there is (i) a data structure that is constructed from the input and (ii) a function which traverses the data structure and performs a computation on its nodes. Frequently, the computations done by the function on different parts of the data structure are *independent*. In the absence of a parallelization mechanism, the computations on different parts of the data structure are done sequentially using *recursive* function calls or *while* loops. During the computation of a subproblem, the function finds new subproblems and continues its execution by moving on to these subproblems. However, if the function has access to the subproblems before starting the initial computation, it can potentially initiate processing of different parts of the problem independently in parallel. To achieve this, before starting the computation on the data structure, our approach invokes a *helper thread* which goes over the data structure and finds multiple independent subproblems in the data structure. It is important to note that the subproblems are runtime entities and cannot be identified at compile time. Then, the function gets access to multiple subproblems and can start working on them in parallel.

The motivation for using a separate thread to identify the subproblems is to reduce the overhead introduced by searching the data structure to identify subproblems. The main thread invokes the helper thread before arriving at the parallel section and proceeds with its work. While the helper thread identifies the subproblems, the main thread can execute the code before the parallel section. Note that, the main thread has to make sure that the target data structure is not modified once the helper thread is invoked until the computation in the parallel section is started. This can be verified using static analysis techniques, or it can also be obtained from the programmer in the form of a directive.

In an on-chip network based CMP with S-NUCA Kim et al. (2003) cache organization, upon an L1 miss, the L1 cache controller issues a request to the L2 cache bank corresponding to the missing address which traverses the on-chip network. If the address misses in the L2 cache as well, the L2 cache controller at that bank issues a request to the corresponding memory controller which again uses the on-chip network. The memory controller determines the memory bank corresponding to the requesting address and places it in the particular bank's queue. When it is time for the memory controller to schedule this request, corresponding memory commands are sent to the memory module (DRAM) over the memory channel. If the memory bank has the requesting address in its *row buffer* (*row buffer hit*), the response will be sent back to the memory controller. Otherwise, the row that is open must be closed and a new row must be activated (*row*

buffer conflict) before the data can be read. This data response travels from the memory controller back to the requesting L2 cache controller through the on-chip network. Finally, the L2 cache controller forwards the response to the original requesting L1 cache.

Workload	**On-chip**	**Off-chip**	
		Queueing	**Access**
High MPKI	18%	60%	22%
Moderate MPKI	35%	19%	46%
Low MPKI	43%	8%	49%

Table 2.1: Breakdown of on-chip and off-chip latencies for L2 misses in three different workloads without any prefetching.

In a CMP system running multiple applications/threads, memory is a shared resource with high contention, which makes it one of the most important performance bottlenecks. Within a short period of execution, multiple L2 misses can happen and those requests will be sent to the memory controllers. To leverage memory level parallelism, multiple memory controllers and multiple banks at each memory controller are deployed. Still, any core in the system can access any bank at any memory controller. This flexibility leads to *inter-core interference* at the memory controllers. This contention/interference in the memory system can lead to significant queueing delays, thereby reducing overall memory system performance.

Table 2.1 shows the breakdown of the average round-trip latency incurred by a memory request in a 32-core CMP with 4 memory controllers in three different multiprogrammed workloads, prioritized from high (>10) MPKI (Misses Per Kilo Instructions) to low(<1) MPKI. In high and moderate MPKI workloads, the total off-chip latencies clearly dominate, with a bulk of these latencies coming from the queueing delays at the memory controller in the high MPKI case. Note that while the DRAM access itself contributes directly to the latency of a memory request, it also indirectly manifests in the queueing delays of other requests waiting at the memory controller. Bringing data on-chip (as is done in our proposal) can reduce such off-chip accesses, which is especially useful in high MPKI applications. At the same time, note that in low MPKI workloads, where on-chip latencies become comparable to off-chip latencies, it is equally important to ensure that we do not add to the on-chip latencies when optimizing the off-chip costs. These contentions at the on-chip network and off-chip memory channels are becoming more prominent with increasing number of cores on chip and get further accentuated in large-scale CMPs if we simply introduce a conventional, core-side prefetcher. Core-side prefetcher can cause significant queueing latency increase in both on-chip network and off-chip memory channels.

These observations motivate our proposal for a memory-side prefetcher which attempts to bring some data from the currently active row in the DRAM on-chip to reduce the off-chip costs for accessing this data while leveraging the fact that proactively reading such data from the row buffer does not incur DRAM row switch out/in costs. Note

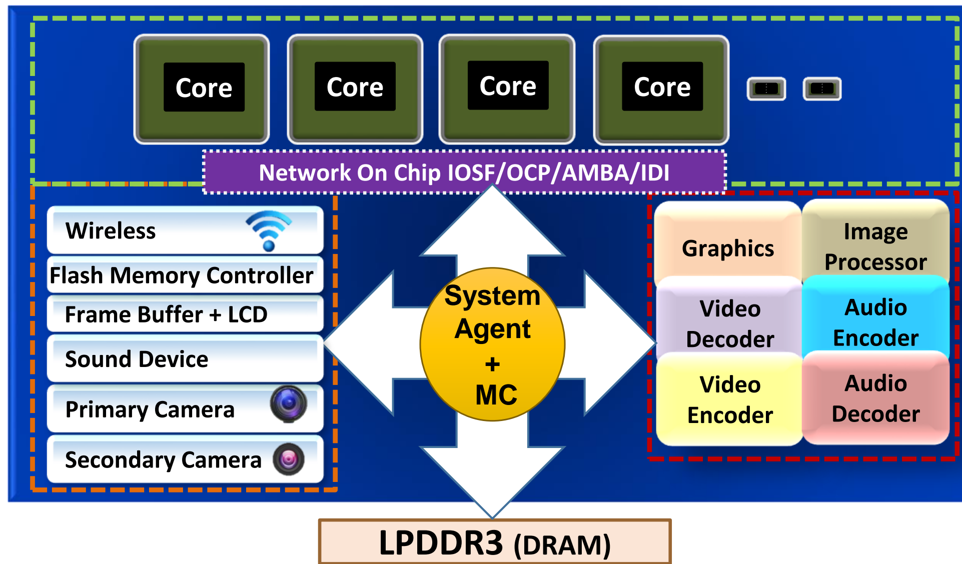


Fig. 2.1: Target SoC platform with a high-level view of different functional blocks in the system.

that such prefetching does *not* need to push/propagate this data to the caches/cores, thus avoiding additional on-chip resource contention which can happen with normal inaccurate and ill-timed core-side prefetches. *Avoiding/reducing off-chip accesses, while not increasing on-chip traffic and not creating cache pollution serves as the primary motivation for our memory-side prefetcher.*

2.1.1 Mobile Platforms

As shown in Figure 2.1, handhelds available in the market have multiple cores and other specialized IPs. The IPs in these platforms can be broadly classified into two categories – *accelerators* and *devices*. Devices interact directly with the user or external world and include cameras, touch screen, speaker and wireless. Accelerators are the on-chip hardware components which specialize in certain activities. They are the workhorses of the SoC as they provide maximum performance and power efficiency, e.g. video encoders/decoders, graphics, imaging and audio engines.

Interactions between Core, IPs and Operating System: SoC applications are highly interactive and involve multiple accelerators and devices to enhance user experience. Using API calls, application requirements get transformed to accelerator requirements through different layers of the OS. Typically, the calls happen through software device drivers in the kernel portion of the OS. These calls decide if, when and for how long the different accelerators get used. The device drivers, which are optimized by the IP vendors, control the functionality and the power states of the accelerators. Once an accelerator needs to be invoked, its device driver is notified with request and associated physical address of input data. The device driver sets up the different activities that the accelerator needs to do, including writing appropriate registers with pointers to

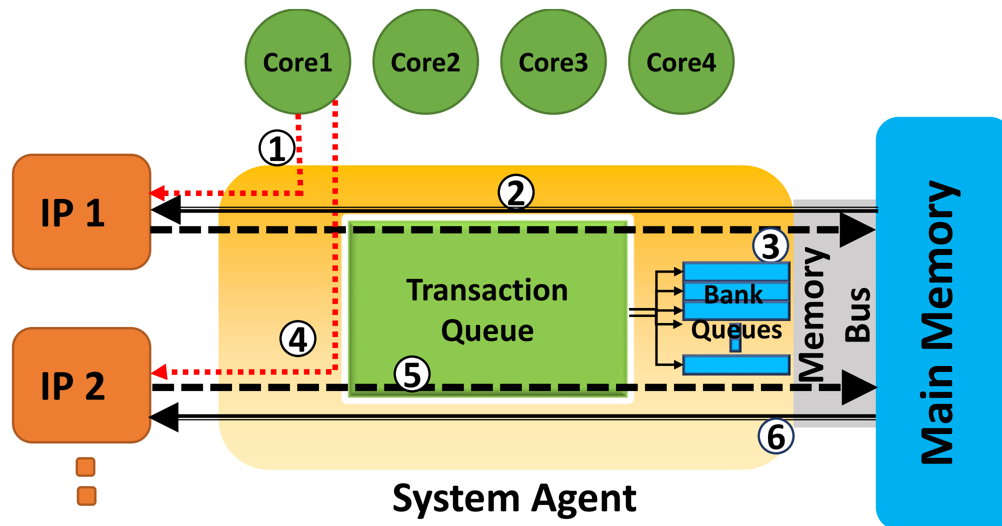


Fig. 2.2: Overview of data flow in SoC architectures.

the memory region where the data should be fetched and written back. The accelerator reads the data from main memory through DMA. Input data fetching and processing are pipelined and the fetching granularity depends on how the local buffer is designed. Once data is processed, it is written back to the local buffers and eventually to the main memory at the address region specified by the driver. As most accelerators work faster than main memory, there is a need for input and output buffers.

The System Agent (SA): Also known as the Northbridge, is a controller that receives commands from the core and passes them on to the IPs. Some designs add more intelligence to the SA to prioritize and reorder requests to meet QoS deadlines and to improve DRAM hits. SA usually incorporates the memory controller (MC) as well. Apart from re-ordering requests across components to meet QoS guarantees, even fine-grained re-ordering among IP's requests can be done to maximize DRAM bandwidth and bus-utilization. With increasing user demands from handhelds the number of accelerators and their speeds keep increasing Steve Scheirey (2013); Engwell (2013); Engwell. These trends will place a very high demand on DRAM traffic. Consequently, unless we design a sophisticated SA that can handle the increased amount of traffic, the improvement in accelerators' performance will not end in improved user experience.

Data movement in SoCs

Figure 2.2 depicts the high-level view of the data flow in SoC architectures. Once a core issues a request to an IP through he SA (shown as (1), the IP starts its work by injecting a memory request into SA. First, the request traverses through an interconnect which is typically a bus or cross-bar, and is enqueued in a memory transaction queue. Here, requests can be reordered by the SA according to individual IP priorities to help requests meet their deadlines. Subsequently, requests are placed in the bank-queues of the memory controller, where requests from IPs are re-arranged to maximize the bus utilization (and in turn, the DRAM bandwidth). Following that, an off-chip DRAM access is made. The response is returned to the IP through the response network in the

SA (shown as (2)). IP-1 writes its output data to memory (shown in (3)) till it completes processing the whole frame. After IP-1 completes processing, IP-2 is invoked by the core (shown as 4), and data flow similar to what IP-1 had is followed, as captured by (5) and (6) in Figure 2.2. The unit of data processing in media and gaming IPs (including audio, video and graphics) is a **frame**, which carries information about the image or pixels or audio delivered to the user. *Typically a high frame drop rate corresponds to a deterioration in user-experience.*

2.2 Related Work

Memory Parallelism vs. Locality There have been various works in the past focusing on either parallelism or locality or both in different levels of the memory hierarchy separately. Kim et al. (2002) proposed static and dynamic cache organizations to improve parallelism and locality in the on-chip cache banks. The proposed Static-NUCA organization optimizes the parallelism in the on-chip cache while Dynamic-NUCA optimizes the locality. The work in Cade and Qasem (2009) explored the parallelism and locality tradeoffs in the context of an on-chip cache in a CMP by manipulating the thread-synchronization points. Huh et al. (2007) explored a spectrum of on-chip cache organizations to find the optimal degree of sharing of cache banks, and Blelloch et al. (2013) presented a program-centric model to improve the locality in on-chip caches. Singhai et al. (1997) presented a loop fusion algorithm to improve the parallelism and locality in the on-chip caches. Mutlu et al. (2003) proposed using run-ahead execution to improve the memory parallelism at the core which leads to better utilization of the on-chip caches and memory subsystem. A recent technique called subarray-level parallelism Kim et al. (2012) manages the subarrays within a memory bank individually, thereby providing parallelism within the memory banks. The work presented in Jeong et al. (2012a) proposed partitioning the memory banks across cores in order to improve the memory bank locality by eliminating inter-thread interference. Another work that targets memory locality is micro-pages Sudan et al. (2010), which puts heavily-accessed parts of different pages into the rowbuffer. By placing parts of multiple pages in the rowbuffer at the same time they showed significant improvements in rowbuffer locality. All the above works focus on only one level in the memory hierarchy either on-chip cache or memory but not both. In contrast, this dissertation presents a comprehensive study on parallelism-locality tradeoffs in all the levels of memory hierarchy including the cross-level interactions.

Memory Migration There have been prior works that explored data migration in the context of CMPs. Awasthi et al. (2010) proposed a data migration scheme that migrates pages between memory channels to balance the row buffer locality at different memory channels. Page-NUCA Chaudhuri (2009) deals with migration of pages from one cache bank to another to improve cache locality. They bring heavily accessed pages to home L2 banks, thereby reducing the hit latency. Eisley et al. (2008) presented a cache line migration scheme, where a cache line upon eviction from a cache bank is migrated to a nearby cache bank. The on-chip routers keep track of cache utilization of the neighboring nodes to route the evicted cache line to the under-utilized node. Most of these migration techniques optimize for locality without much regard for parallelism

which is also important as shown in our characterization study. In contrast, our migration scheme optimizes both the parallelism and locality at the same time. Moreover, in the above schemes, the primary method of handling migration overheads is by being very selective on which data to migrate. The migration mechanisms proposed in this work are generic and can be employed for any migration policy.

Data Reuse: Data reuse within and across cores has been studied by many works. Chen et al. (2005); Xue et al. (2006), Gordon et al. (2002) and Kandemir et al. (2002) propose compiler optimizations that perform code restructuring and enable data sharing across processors. Suhendra et al. (2006) proposed ways to optimally use scratch pad memory in MPSoCs along with methods to schedule processes to cores. There have been multiple works that discuss application and task mapping to MPSoCs Marwedel et al. (2011); Singh et al. (2010); Coskun et al. (2007) with the goal of minimizing data movement across cores. Our work looks at accelerator traffic, which is dominant in SoCs, and identifies that frame data is reused across IPs. Unlike core traffic, the reuse can be exploited only if the data frames are broken in sub-frames. We capture this for data frames of different classes of applications (audio/video/graphics) and propose techniques to reduce the data movement by short circuiting the producer writes to the consumer reads.

Memory Controller Design: A large body of works exist in the area of memory scheduling techniques and memory controller designs in the context of MPSoCs. Lee and Chang Lee and Chang (2006) describe the essential issues in memory system design for SoCs. Akesson et al. (2007) propose a memory scheduling technique that provides a guaranteed minimum bandwidth and maximum latency bound to IPs. Jeong et al. (2012b) provide QoS guarantees to frames by balancing memory requests at the memory controller. Our work identifies a specific characteristic (reuse at sub-frame level) that exists when data flows through accelerators and optimizes system agent design. Our solution is complimentary to prior techniques and can work in tandem with them.

Along with IP design and analysis, several works have proposed IP-specific optimizations Khan and Anwar; Fenney (2003); Shim et al. (2004); Han et al. (2013) and low power aspects of system-on-chip architectures Gutierrez et al. (2011); Wang et al. (2010); Diniz et al. (2007). Our solution is not specific to any IP rather, it is at the system-level. By reducing the IP stall times and memory traffic, we make the SoC performance and power-efficient.

Prefetching: Hughes and Adve (2005) proposed the use of memory prefetching to improve the performance of programs working with linked data structures. They exploit the predictability of accesses available in linked data structures to do the prefetching. Their work focuses on a specific kind of programs whereas the idea presented in this proposal in more versatile and works with any kind of programs. The work in Hur and Lin (2006) proposed an Adaptive Stream Detection (ASD) technique at the memory controller to identify the short-streams. They control the aggressiveness of the stream prefetcher using Stream Length Histogram (SLH) that are computed periodically. Their technique could stop useless prefetches. As indicated previously it is mainly a prediction technique which dictates *what to prefetch?* while ours is a complete memory prefetching solution which can actually incorporate their prediction technique. Impulse Carter et al. (1999) proposed the use of a smarter memory controller which does address remapping

and prefetching for better performance. Both these schemes do not take into account the state of the channel or the row buffers before prefetching. In contrast, the work presented in this dissertation considers the current memory state before issuing the prefetches so as to minimize the overhead of prefetching.

The work presented in Lin (2001) proposed the use of prefetchers both in the L2 caches and the memory controllers to improve performance. They identify idle cycles on the memory channel and utilize them to schedule the prefetch requests. The work in Ortega et al. (2002) proposed a hybrid software/hardware prefetch scheme. In their scheme, hardware prefetcher does prefetching into the L1 cache, while the compiler-directed special load and prefetch instructions bypass the data into the registers directly. The work presented in Joseph and Grunwald (1997) uses Markov Predictors for memory-side prefetching in a uniprocessor system. They also proposed the use of on-chip prefetch buffers along with L1 caches. Solihin et al. (2003) proposed a user-level memory thread based software prefetching. This user-level memory thread runs on a general purpose core present on the memory controller chip. All these techniques are unicore based and employ a push-based strategy where the prefetched data is pushed to the on-chip entities which leads to network congestion and cache pollution in a large-scale on-chip network based CMP.

Automatic Parallelization: Ryoo et al. (2007) point out the importance of automatic parallelization and summarize various sophisticated analyses techniques required from a compiler to parallelize programs with pointers. Tournavitis et al. (2009) propose a profile-driven approach to detect parallelism, instead of limited static analysis techniques. They rely on the user for final approval of the potential transformation. Decoupled software pipelining Ottoni et al. (2005) and parallel stage decoupled software pipelining Raman et al. (2008a) (DSWP and PS-DSWP) represent non-speculative approaches to automatic parallelization by pipelining the loop iterations. Vachharajani et al. (2007) explored an extension of DSWP by adding speculation in order to achieve better speedups.

Another approach towards automatic parallelization is using commutativity analysis Aleen and Clark (2009); Rinard and Diniz (1996). It is based on the idea that, if the program states reached after executing two statements in any order are equivalent, then these statements are said be commutative. Since the order in which these statements execute does not matter, they can be executed in parallel.

Rus et al. (2007, 2003) introduced a technique for automatic parallelization by combining static and dynamic analyses of programs. In their work, the authors extract conditions for parallel execution by static analysis and use them to guard the dynamic parallelization of loops at runtime. The LRPD Test in Rauchwerger and Padua (1995a) and the R-LRPD Test in Dang et al. (2002) are techniques for automatic parallelization using only runtime analysis. In these works, target loops are executed speculatively in parallel and tested for memory dependences at runtime.

Most of the prior works in this topic focus on using runtime information to parallelize array based applications. The work presented in this proposal, focuses on parallelizing irregular programs where runtime information is much more important.

Bridges et al. (2007) presented a framework that brings together techniques from compiler and hardware domains. They also indicate that, by extending the programming languages, programmers can give hints to the compiler about potential parallelism opportunities. OpenMP Standard (2009) is a successful example of programmer driven parallelism. It takes directives from the programmers which express parallelism in the code and executes the code in parallel. Grant et al. (2000) used annotations in programs written using a declarative language to dynamically compile and specialize the program to the actual running environment.

Rauchwerger and Padua (1995b) presented a framework to parallelize while loops. They evaluate the recurrences that can be statically identified in parallel and speculatively execute the remainder of the loop concurrently. In their approach, they have to undo the effects of any iterations that overshoot the termination condition. We do not perform any speculative computations, and there is no need to undo computations in our approach. Gupta et al. (2000) presented a technique to parallelize recursive procedures that typically appear in divide-and-conquer algorithms. They discuss a compile time analysis technique which works good for array based programs and use speculation to run the code. Rogers et al. (1995) presented a execution model for supporting programs that use pointer based dynamic data structures on distributed memory systems. They address the issues of data placement in such applications on a distributed memory platform. The work presented here targets shared memory machines where data layout and data migration issues can be safely ignored.

There have been several efforts to analyze pointers at compile time. Rugina and Rinard Rugina and Rinard (2003) present an approach to pointer analysis in multithreaded programs. They compute a conservative approximation of memory locations to which each point may point to. Guo et al. (2005) propose the technique of performing pointer analysis on a low-level intermediate representation obtained after the code transformations. They compare their technique to propagating high-level pointer analysis information through subsequent code transformations. In Da Silva and Steffan (2006), the authors present a probabilistic pointer analysis approach that statically predicts the probability of each points-to relation. This information is used in applying various speculative optimizations.

There have been two approaches in the literature that employ speculation in the context of code parallelization. The first approach, namely, memory alias speculation Steffan and Mowry (1998); Bruening et al. (1998); Du et al. (2004); Zhong et al. (2008), assumes that the addresses accessed by different threads do not overlap. In Du et al. (2004), a cost driven compilation framework for speculative parallelization is presented. In this framework, consecutive iterations of a loop are executed in parallel by speculating that the potential data dependencies across the iterations do not materialize at runtime. Speculative code transformations to enhance thread level parallelism are presented in Zhong et al. (2008). The work by Bruening et al. (1998) is based on the observation that the memory access patterns can often be predicted at runtime using simple value predictors.

The second approach to use speculation for parallelization is value speculation Raman et al. (2008b); Quińones et al. (2005), which predict the values needed in future iterations and execute these iterations in parallel speculatively. In Quińones et al. (2005),

the most effective points in a program to spawn speculative threads are identified. The work by Raman et al. (2008b) distribute chunks of iterations to threads, and as a result, reduce the number of predictions needed. Another approach to thread level speculation for automatic parallelization Steffan et al. (2000); Steffan and Mowry (1998) is through hardware support in the form of additional cache states.

It should be noted that speculative parallelization may suffer from a high mis-speculation rate in some cases, which can result in the parallel code performing worse than the sequential code. A high mis-speculation rate may also result in high power consumption, since in the case of a mis-speculation, all the incorrect (speculative) results are flushed and the code is executed sequentially resulting in the wastage of all the speculative computations. Furthermore, as stated in Du et al. (2004), in speculative parallelism, for the speculative threads, all the speculative results (including memory writes) are buffered and are not part of the program state. As a result, even after the speculated condition is discovered to be correct, a separate stage to update the program state with the speculative results in the buffer is required.

The work presented in this proposal is a non-speculative approach to parallelization. There is no need to maintain a different program state and update it in our approach. Moreover, we propose techniques to totally hide the performance overheads incurred in our approach. By employing these techniques, the parallel code generated from our approach never performs worse than the sequential version.

Chapter 3

Memory Migration

3.1 Parallelism vs. Locality

Memory subsystem performance is a primary determinant of application performance. In a typical system, L1 cache is organized as private to optimize for access latency and there is not much scope for parallelism or locality at this level. Starting from the L1 miss, each level in the memory hierarchy exhibits different tradeoffs in parallelism and locality which impact the L1 miss latencies. The parallelism and locality characteristics are heavily influenced by the address mapping employed. This section characterizes the tradeoffs of different address mappings in the memory hierarchy.

3.1.1 Mapping Scenarios

In an NoC based CMP with multiple L2 banks, there are a multitude of ways to map the physical address space to the L2 banks. On one end of the spectrum, the L2 cache is privately mapped where full address space is mapped to each and every L2 bank and each bank holds the data related only to the corresponding core. On the other end, a shared L2 cache can be organized as an S-NUCA Kim et al. (2003) system where parts of the address space are striped across different L2 banks. Two popular ways of striping the data are *line*-level striping where consecutive cache lines are mapped to different L2 banks in a block cyclic manner and *page*-level striping where consecutive pages are mapped to different L2 banks.

Going to the next level in memory hierarchy, the address space can be mapped across the memory channels as private or line-level or page-level striping. Similarly, within each memory channel the addresses can be mapped to different memory banks in those three manners. Table 3.1 gives a list of the scenarios we evaluated in our characterization. To keep the number of simulations manageable we evaluated only line-level and page-level stripings at the memory banks. Note that, in the cases with line-level striping at the memory channel, the pages are already distributed across different channels, thus it is not possible to have page-level striping at the memory banks. In the following subsections we thoroughly explore the tradeoffs caused by these address mappings at each level in the memory hierarchy.

3.1.2 L2 Cache

As discussed earlier, an L1 miss can have different behavior depending on whether it (i) hits in the L2 in which case response is sent back, or (ii) misses in the L2 cache in which case the request has to be forwarded to the memory. In either case every L1 miss has to access an L2 bank (local or remote). From an L1's perspective, the performance of

Serial No.	L2 Bank	Memory Channel	Memory Bank
1	Line	Line	Line
2	Line	Page	Line
3	Line	Page	Page
4	Line	Private	Line
5	Line	Private	Page
6	Page	Line	Line
7	Page	Page	Line
8	Page	Page	Page
9	Page	Private	Line
10	Page	Private	Page
11	Private	Line	Line
12	Private	Page	Line
13	Private	Page	Page
14	Private	Private	Line
15	Private	Private	Page

Table 3.1: Evaluated mapping scenarios.

L2 cache is can be viewed in two dimensions: L2 parallelism and L2 locality. Parallelism is a measure of how many L2 banks are serving an L1 cache in parallel, and locality is a measure of how far the request has to travel on-chip in order to access the L2 bank. Both of these metrics play an important role in determining the time spent in servicing L1 cache misses.

We define *L2 parallelism* as the number of L2 banks serving an L1 in a small epoch of 128 cycles (calculated based on the processor ROB size). Typically, an L1 can have multiple outstanding requests to the L2 cache at a time. If these requests are distributed across different L2 banks, then the utilization of L2 banks and network resources tends to be uniform. The more number of L2 banks serving an L1, the more parallelism, leading to faster L1 miss response times. *L2 locality* can be defined as the time spent by an L1 miss request in the on-chip network to reach the corresponding L2 bank. This is primarily determined by the distance between L1 cache and the L2 bank in the on-chip network. Shorter distances mean better *L2 locality*, leading to faster L1 miss response times. Different L2 mappings (line-level, page-level and private) exhibit very different locality and parallelism characteristics. A line-level striping at L2 distributes the address space at a very fine granularity (cache-line) leading to better L2 parallelism although at the cost of L2 locality. On the other end, private organization of L2 banks is optimal from the locality perspective although at the cost of L2 parallelism. In the middle of the spectrum is page-level striping which has better parallelism than the private L2 organization and better locality than the line-level striping.

For clarity, in the following graphs, each workload category is represented as one bar which is the *average* of values obtained in the four (very similar) workloads under that category. When analyzing tradeoffs at a level in the memory hierarchy, the mappings at the next level do not matter and consequently those combinations are not shown in the following graphs.

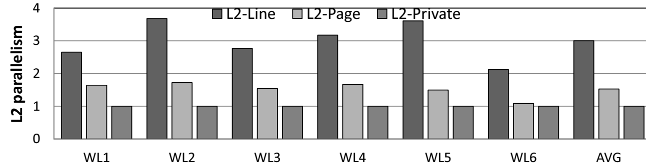


Fig. 3.1: L2 parallelism under different mappings. (Higher is better)

Figure 3.1 shows the *L2 parallelism* metric across different workloads under different L2 mappings. The memory channel and banks are set to line-level striping for these experiments. It can be seen that line-level striping provides high L2 parallelism compared to other two schemes, and private L2 organization has the least parallelism; in fact, it is always 1. As expected page-level striping stands in between the two mappings.

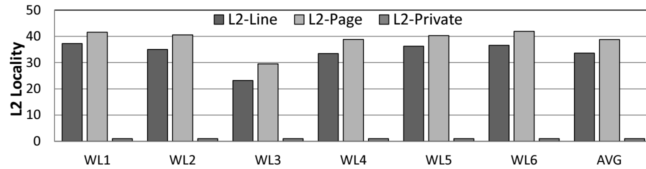


Fig. 3.2: L2 Locality under different mappings. (Lower is better)

Figure 3.2 shows the *L2 locality* metric across different workloads under different L2 mappings. Private L2 organization gives the best locality (always local). Line-level striping has slightly better L2 locality than page-level striping.

Figure 3.3 shows the *Normalized IPC* under different L2 mappings normalized to line-level striping. Because of the loss in L2 hit rate , page-level striping could not perform on par with the line-level striping. Private L2 organization also performed worse than the line-level striping in all the cases except low MPKI workloads (WL3).

From the above analysis it can be concluded that, parallelism is more important than locality at the L2 cache level, because of the need to serve a burst of L2 requests in a short time. Having a shared L2 cache is desirable to multiplex the cache capacity and support demand from multiple cores, and an S-NUCA with line-level striping gives the best parallelism at L2 level leading to better performance.

3.1.3 Memory Channel

An L2 miss is forwarded to the corresponding memory controller determined by the address mapping at the memory channel level. Memory parallelism is a measure of how many memory channels are serving the L2 cache at a time, and memory locality is a measure of how far a request has to travel in order to reach the corresponding memory controller. More specifically, *memory channel parallelism* is defined as the number of

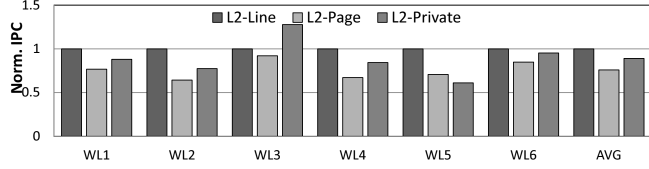


Fig. 3.3: Normalized IPC under different mappings.

memory channels serving the L2 cache (misses) in a small epoch (128 cycles [1]). An L2 can have multiple outstanding requests to the memory at a time. If these memory requests utilize different memory channels, individual memory controllers will not be overwhelmed with requests. The more number of memory channels serving an L2, the more parallelism, leading to better L2 miss response times. *Memory channel (MC) locality* is defined as the time spent by an L2 miss in the on-chip network to reach the corresponding memory controller. This is primarily determined by the distance between the L2 bank and the memory controller. Shorter distances mean better memory locality, leading to faster L2 miss response times. Similar to the L2 mapping, address space can be mapped to the memory channels in the system in three ways (i) private, (ii) line-level striping, and (iii) page-level striping. In the private memory channel organization, each memory channel caters to a set of cores in the system. All L2 misses from an application are served by a particular memory controller. While we assigned private memory channels to cores, it is also possible to assign the channels to a set of L2 banks. However, we selected the former definition because such an organization has the potential to improve locality inside the memory (explained in the next subsection). A line-level striping, being fine grained, provides high parallelism at the memory channel level. Page-level mapping is a compromise between the two mappings in terms of both parallelism and locality.

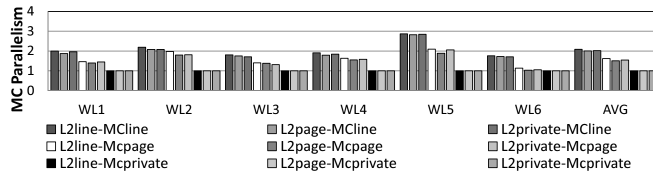


Fig. 3.4: Memory channel parallelism under different mappings. (Higher is better.)

Figure 3.4 shows the *memory channel parallelism* metric across different workloads under different L2 and memory channel mappings. The memory banks are set to line-level striping for these experiments. It can be seen that line-level striping provides the highest memory channel parallelism; private memory channels give the least parallelism;

[1] At an epoch length of 128 cycles, noticeable differences in memory characteristics are observed. Larger epochs tend to obscure such details.

and, page-level striping stands in between the two schemes. Within a memory channel mapping there is no significant effect of different L2 level mappings on memory channel parallelism.

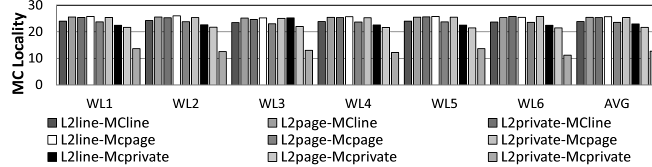


Fig. 3.5: Memory channel locality under different mappings. (Lower is better.)

Figure 3.5 shows the *memory channel locality* metric across different workloads under different L2 and memory channel mappings. In general, there is not much difference between the memory channel localities observed in different address mappings except for the case of private memory channel with private L2 (*L2private-MCprivate*). *L2private* means the core to L2 mapping is fixed; *MCprivate* means core to memory channel mapping is fixed. When we combine both of them, the L2 to memory channel mappings are fixed and consequently every L2 bank always accesses the nearest memory controller, thereby improving the memory channel locality significantly.

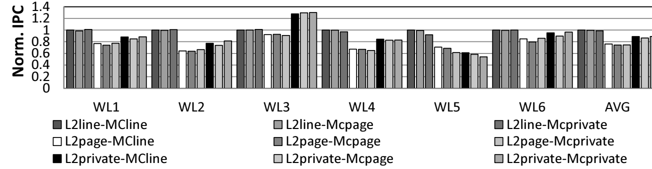


Fig. 3.6: Normalized IPC under different mappings.

Figure 3.6 shows the *Normalized IPC* under different memory channel mappings normalized to L2 line-level and MC line-level striping (first bar). Under a given L2 mapping there is no significant difference in IPC caused by different memory controller mappings.

3.1.4 Memory Bank

Once a memory request is received at a memory controller, relevant memory commands have to be sent to the memory bank corresponding to the requesting address. The parallelism and locality at the memory banks impact the performance significantly. *Memory bank parallelism* is defined as the number of memory banks busy at the memory channel when at least one memory request is being serviced at that channel. Higher memory bank parallelism means better utilization of memory banks leading to shorter queueing delays at the memory controllers. *Memory bank locality* is defined as the average

row buffer hit rate at the memory banks. Higher memory bank locality means more row buffer hits leading to shorter memory access delays. We explored two ways of address mapping at the memory banks (i) line-level striping and (ii) page-level striping. Line-level striping being very fine granular provides high parallelism at the memory banks, whereas page-level mapping provides high locality in the memory banks because a whole page is mapped to a bank and its row buffer leading to more row buffer hits.

In general, line-level striping at the memory banks gives better memory bank parallelism compared to page-level striping. Within the line-level striping at the bank, the mappings with line-level striping at L2 showed slightly lower memory bank parallelism than the other two options. Page-level striping at the memory banks gives significantly better memory bank locality compared to line-level striping. As mentioned above, memory bank parallelism yields shorter memory queueing delays. In general line-level striping at the memory banks show shorter memory queueing delays compared to the page-level striping.

Memory bank locality yields shorter memory access latencies. Page-level striping at the memory banks show shorter memory access latencies compared to line-level striping because of memory bank locality.

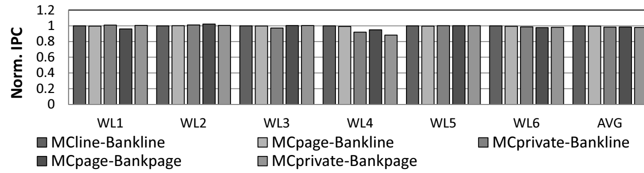


Fig. 3.7: Normalized IPC under different memory mappings.

In order to clearly show the impact of memory bank mapping on performance, we selected the line-level mapping at L2. Figure 3.7 shows the *Normalized IPC* under different memory bank mappings normalized to line-level mapping at channel and bank (first bar). The first three bars show the normalized IPC with line-level mapping at the memory banks, and the last two bars show the normalized IPC with page-level mapping at the memory banks. From the graph, it can be seen that there is no significant impact of memory bank mapping on the IPC. However, different address mappings exhibit very different memory queueing delays and access latencies. This is caused by the loss in locality in the case of line-level mapping although it has good parallelism. In the same way, page-level mapping has good locality but less parallelism leading to similar performance in both the cases.

3.1.5 Summary of Findings

Address mapping plays an important role in determining the parallelism and locality in the memory hierarchy, which in turn determine the memory subsystem performance. We explored different address mappings at the L2 cache, memory channel

and memory bank levels ranging from line-level to page-level to private mapping. The conclusions from our analysis are as follows:

- At the L2 level, parallelism is more important than the locality. Having a shared L2 cache is desirable to support demand from multiple cores in parallel. An S-NUCA with line-level mapping gives the best parallelism at L2 leading to the best performance.

- At the memory channel and bank levels on the other hand, there is no clear winner on which is more important parallelism or locality. Parallelism reduces the queueing delays at the memory while locality reduces the memory access latencies. Although, all the mappings tried at the memory level delivered fairly similar performance, the means to achieve it were different – through parallelism *or* locality.

Through the findings of above analysis, we develop a technique which exploits both – parallelism *and* locality in the memory subsystem, for better system performance.

3.2 Dynamic Migration for Improving Memory Performance

The characterization analysis presented in the previous section can be used in multiple ways. By controlling the parallelism and locality in the memory hierarchy one can improve the system performance or save power or design better prefetching schemes. In this section, we explore a novel data migration scheme which exploits the parallelism and locality in the memory subsystem to improve system performance.

A key observation from our characterization study is that different address mappings lead to different performance characteristics. For example, a page-level striping at the memory banks yields better locality leading to shorter memory access latencies, whereas a line-level striping yields better parallelism leading to shorter memory queueing delays. Based on the application's memory access patterns, some applications might benefit from better parallelism from the memory subsystem and others prefer better locality. In fact, within an application, there could be parts of data that prefer parallelism, while other parts prefer locality. These scenarios can be supported by a heterogeneous memory organization where some data can be striped at page-level and other striped at line-level. We present a dynamic migration scheme which identifies the ideal striping level for each page in the memory and migrates the data to realize that striping. Obtaining this information by profiling the application is not feasible because it is very sensitive to the program input and operating system memory management.

3.2.1 Migration Policy

The most important question in any migration scheme is *Which data should be migrated?* This can be answered by identifying which pages in memory prefer parallelism and which pages prefer locality. We propose the use of a runtime monitoring system to identify such information. In a memory system with page-level striping at the memory channels and memory banks, all the requests to a page are mapped to a particular

bank. For pages that are accessed in bursts, multiple requests are sent to the memory in a short period of time (like 128 cycles) which can lead to substantial queuing at the memory bank. If the same page is mapped with line-level striping, these requests are sent to different memory controllers and memory banks resulting in less queueing. In the same manner, in a memory system with line-level striping at the memory channels, the locality of data is significantly reduced. For pages that exhibit high locality in accesses, it is preferable to map them with page-level striping to exploit the row buffer locality. Based on these observations, we propose a dynamic migration scheme which supports heterogeneous address mapping at the memory. The system starts with a homogeneous memory organization where all the data is mapped as either line-level striping or page-level striping. While the application is running, a runtime system monitors the accesses to different pages in the memory and decides the best mapping strategy for each page. If a page gets multiple requests in a short period of time, line-level striping is preferable for that page. Similarly, if a page exhibits high spatial locality, page-level striping is preferred for that page. At the end of each epoch (10,000 cycles) the pages are either distributed across multiple channels/banks (for line-level striping) or consolidated at one bank (for page-level striping) according to the mapping identified for that page. This can lead to better parallelism in a page-level striped system without much loss of locality in the memory subsystem, and to better locality in a line-level striped system without much loss of parallelism. The striping information for the pages can be stored using one bit either in the page table or at the L2 banks.

3.2.2 Migration Mechanisms

Once the data to be migrated is identified, the next question to be answered is *How to efficiently migrate the data?* Although migration has the potential to reduce the memory access times, it comes with a significant overhead. There can be large amounts of data that have to be migrated across the memory channels or banks.

One can build a data migration scheme which restricts the migrations to be within the memory channel. Such a migration scheme does not incur the *data transfer* overheads and the data can be migrated using techniques like *rowclone* Seshadri et al. (2013). However, those schemes cannot unlock the full parallelism and locality benefits provided by an unrestricted scheme which improves the memory channel parallelism also. Therefore, the rest of this section focuses on how to handle the overheads incurred in a full migration scheme. We present two novel migration mechanisms to reduce these overheads, thereby allowing significantly more number of page migrations.

Smart Eviction: One way to reduce the overhead of migration is by reducing the amount of data that has to be migrated. The key observation here is that parts of the page that has to be migrated might already be present in the L2 cache on-chip. We can utilize this data already on the chip to reduce the amount of data read from the memory thereby reducing the data access overhead.

Figure 3.8 shows the percentage of the data that has to be migrated already present in the L2 cache. It can be seen that on average 17% of the data to be migrated is already in the L2 cache and do not have to be read from memory. We leverage this by directly transferring the data present in L2 cache from the L2 bank to the *recipient*

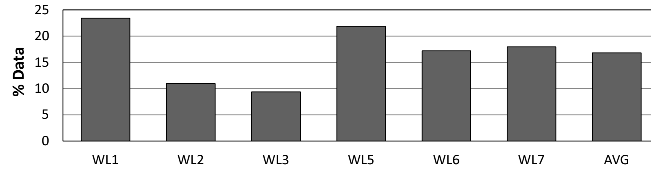


Fig. 3.8: Percentage of data to be migrated already present in the L2 cache.

MC and transferring the remaining data from the *donor MC* directly to the *recipient MC* over the on-chip network.

Separate Network The main problem with using the on-chip network for data migration is the bursty nature of the migration traffic. By injecting such a traffic to the on-chip network, the network can go into saturation at times and affect the whole system performance. We propose the use of a separate network to handle the migration traffic. There have been efforts in the past Mishra et al. (2013); Das et al. (2013) that use multiple networks on chip for different applications. In this work, we use the extra network for migration traffic between memory controllers. Accordingly, it is only connected to the memory controllers and is isolated from the original on-chip network. This network can be used for other purposes as well in the future, but in this work we dedicate it to data migrations.

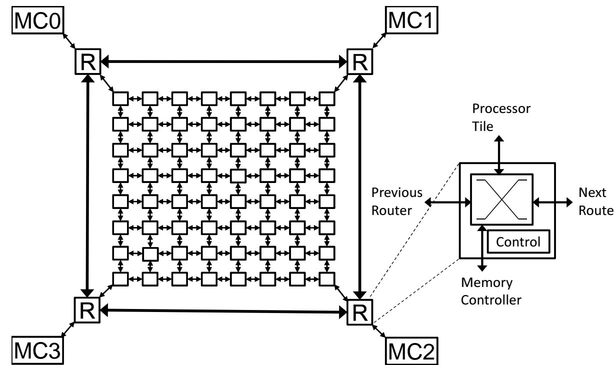


Fig. 3.9: Ring network connecting the memory controllers is used for data migration.

Figure 3.9 shows the ring type of network connecting the memory controllers in a chip. The memory controller is now connected to the processor through a new router ("R"). The router on the ring has four ports, one connected to the memory controller, another connected to the main router in the processor tile and the remaining two to connect to the previous and next routers on the ring. For simplicity and compatibility reasons, we assume this network runs at the same frequency and has the same link width (128 bits) as the original on-chip network. The router forwards the normal request and response packets between the memory controller and the on-chip router. When a migration packet is received from the memory controller, the router forwards it to

the corresponding link on the ring network based on the destination memory controller. When a migration packet is received from the ring network, depending on the destination it is either sent to the connected memory controller or forwarded to the next router in the ring network.

Hybrid Smart eviction technique mainly reduces the data access overheads, while a separate network targets the data transfer overheads. The two techniques can be combined to develop a migration mechanism which has advantages of both the mechanisms. In the combined scheme, as in the smart eviction technique, part of the data is transferred from the L2 bank to the *recipient MC*. The remaining data is read from the *donor MC* and transferred to the *recipient MC* via the separate ring network.

3.3 Experimental Evaluation

In this section, we present the results obtained with our data migration scheme. There are two ways to realize our migration idea (i) starting with page-level striping and trying to increase the memory parallelism by distributing the data or (ii) starting with line-level striping and increasing memory locality by consolidating the data. The following evaluation is for the first approach. Similar results are expected in the second approach also.

3.3.1 Migration Policy

We compare our new migration policy to the one proposed in Awasthi et al. (2010) which migrates the data in order to balance the row-buffer locality at different memory controllers. Our migration policy changes the address mapping for some pages in memory to optimize the parallelism and locality metrics.

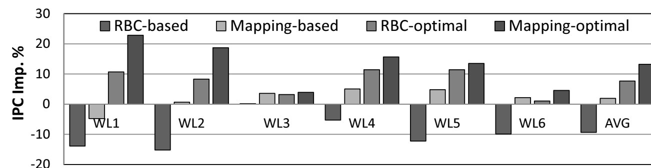


Fig. 3.10: Performance improvements with different migration policies.

Figure 3.10 shows the performance improvements obtained with different migration policies. The y-axis shows the percentage IPC improvements compared to the base case of no migration (page-level striping at memory channels and banks). When the overheads are not modeled, both the policies performed decently yielding average improvements of 7.7% and 13.2% with row-buffer locality and mapping based policies respectively. Note that, the improvements with our mapping based policy are always better than the improvements with the row-buffer locality based policy. However, when the

overheads are accurately modeled, the improvements with row-buffer locality based policy dropped significantly to -9.4%, while our mapping based policy managed to achieve 1.9% improvement. The poor performance of load based policy is due to the overheads in on-chip and off-chip resources which are amplified in the large NOC based CMP.

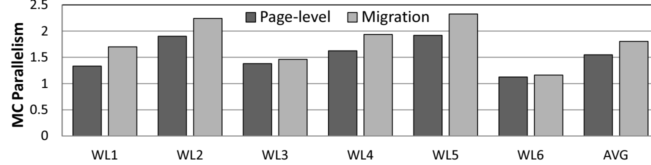


Fig. 3.11: Improvements in memory parallelism obtained by our migration scheme.

Figure 3.11 shows the memory channel parallelism observed with our migration scheme. The first bar for each workload shows the channel parallelism obtained when all the pages are mapped with page-level striping and the second bar shows the channel parallelism with our data migration scheme where selected pages are changed to line-level striping for better parallelism. It can be seen that using our migration policy we are able to improve the memory channel parallelism by 16.1%.

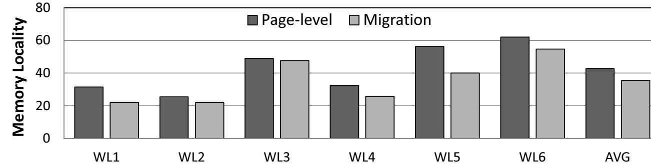


Fig. 3.12: Memory locality with our migration scheme.

Figure 3.12 shows the memory bank locality observed with our migration scheme. By migrating only suitable pages our scheme does not loose much memory locality compared to the page-level striping case. Overall, while our approach is promising from a parallelism and locality perspective, the associated overheads are high and efficient migration mechanisms are needed for better performance improvements.

3.3.2 Migration Mechanisms

The migration mechanisms (*smart eviction*, *separate network*, *hybrid*) described in Section 3.2.2 are evaluated in this section with our mapping-based migration policy. We compare these three mechanisms to a migration that uses on-chip network and an optimal case that does not model any overheads.

Figure 3.13 shows the performance improvements obtained with different migration mechanisms discussed in this dissertation. Both the *data access* and *data transfer* overheads are predominant in the *On-chip* mechanism thereby adversely affecting the

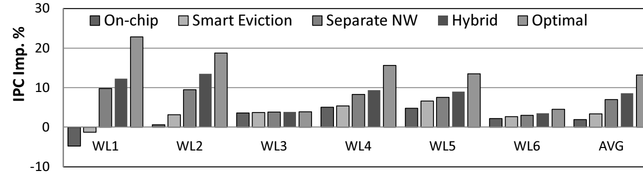


Fig. 3.13: Performance improvements with different migration mechanisms.

performance (1.9% on average). *Smart Eviction* mechanism gives slightly better improvements (3.4% on average) by reducing the *data access* overhead but still incurring the *data transfer* overhead. The *Separate Network* mechanism only reduces the *data transfer* overhead and gives 7% improvement. The hybrid technique reduces both the overheads and yields maximum performance (8.6%), with all overheads included.

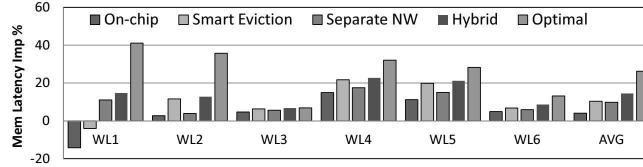


Fig. 3.14: Memory latency with different migration mechanisms.

By improving the memory channel parallelism our migration scheme handles bursts of memory accesses efficiently thereby reducing memory queuing delays. Figure 3.14 shows the reduction in memory latency (queueing delay + access latency) with different migration mechanisms. The proposed *smart eviction* mechanism reduces the *data access* overheads and achieves better memory latencies compared to the other mechanisms.

From the above analysis it can be concluded that the proposed migration policy is effective improving the system performance by reducing the memory latencies. Also, the proposed mechanisms (*smart eviction* and *separate network*) reduce the overheads of data migration, thereby enabling more data migrations. Moreover, the hybrid mechanism combines the benefits of both the mechanisms giving even better results.

Chapter 4

Memory Prefetching

A memory side prefetcher can be developed in the context of large CMPs, as explained below:

• With memory (DRAM) performance becoming a serious bottleneck, understanding and optimizing the requests issued to it based on its current state (especially leveraging the row buffer locality) is becoming extremely critical Cade and Qasem (2009); Sudan et al. (2010); Zhang et al. (2000). A memory-side prefetcher may have more accurate knowledge to optimize such locality (reduce row-buffer conflicts and further augment the effectiveness of memory controller scheduling algorithms Kim et al. (2010c); Mutlu and Moscibroda (2008)), compared to a core-side prefetcher which does not have instantaneous access to memory state.

• While a core-side prefetcher relies on accurate predictions for being effective, a memory-side prefetcher can afford to be more "opportunistic", e.g., initiate certain requests based on bandwidth availability in the memory system. A core-side prefetcher is unaware of the memory status to nimbly adjust its aggressiveness.

• A core-side prefetcher employs *round-trip messages*, and its inaccuracies increase the contention in the on-chip network as will be shown in later. Its purpose is to bring the data into an appropriate cache/buffer so that hit rates can be improved. In a large-scale CMP having dozens of cores, while one would ideally want the data to be present in the caches when requested, *reducing miss latencies can be even more important*. Though one could build a general scheme (as in prior work Iacobovici et al. (2004); Karlsson et al. (2000); Lin (2001); Poulsen and Yew (1994); Smith (1978)) where the data is pushed up to the caches, it does *not* need to be a mandatory requirement for a memory-side prefetcher. A memory-side prefetcher that just brings data on-chip (say to the memory controller), can cut as much as 82% of the total round-trip latency of a normal load request as shown in Table 2.1. Consequently, unlike a core-side prefetcher, a memory-side prefetcher need not contribute to on-chip network contention as show in the next section, while still offering the potential of removing off-chip latency.

• We want to emphasize that a memory-side prefetcher does not preclude the provisioning of a core-side prefetcher as well. In fact, it is quite possible that the two could work in unison (as a decoupled prefetching solution), with the former bringing the data on-chip while leveraging the instantaneous state and capabilities of the memory banks/channels, and the latter leveraging the predictability of core requests to pick up the data (without going off-chip) brought in by the former.

Based on these ideas, we proposes a memory-side prefetcher that is integrated with the memory controllers of an on-chip network based CMP. It maintains a relatively small prefetch buffer of about 256KB per controller, into which our logic prefetches cache lines

from DRAM row buffers based on various factors – predictability, bandwidth availability, row buffer conflict overheads, utility to different request streams, etc. We compare our memory-side prefetcher to a state-of-the-art stream-based core-side prefetcher which employs techniques from Ebrahimi et al. (2011), a next-line core-side prefetcher, a variation of our memory-side prefetcher which pushes the data to caches, and an existing memory-side prefetcher Lin (2001) which pushes the data prefetched at bus idle times to on-chip caches. We show that our memory-side prefetcher outperforms all of those prefetchers. Using both *multiprogrammed* and *multithreaded* workloads, with varying memory pressures, running on a 32-core simulation platform with DDR3 memory, we show that our proposal gives an average IPC improvement of 6.2% (maximum of 33.6%) over no prefetching case when running alone, and 10% (maximum of 49.6%) when combined with a core-side prefetcher. We also perform a sensitivity study to demonstrate the robustness of our memory-side prefetcher in different configurations. Our results show that:

- Core-side prefetching does cut memory access latencies for CMPs, however its effectiveness decreases sharply as the CMPs get larger (like 32 cores).
- Existing memory-side prefetching schemes have better scope in large CMPs, but their effectiveness is limited due to increase in on-chip queueing delays.
- On large-scale CMPs, our mid-way memory-side prefetcher generates better results than conventional core-side and memory-side prefetchers, and complements core-side prefetcher to amplify the benefits.

4.1 Memory-Side Prefetching

In this work, we use memory-side prefetching to improve the latency of off-chip memory requests in large CMPs. Our prefetching scheme improves memory performance in two ways: (i) An access to prefetched (on-chip) data is served faster. (ii) An access to prefetched data will not use the memory bank or the memory channel, which reduces the contention on these highly shared resources, and as a result, reduces the queuing delay for all memory requests. We now explain the details of our proposed prefetching scheme.

4.1.1 What to Prefetch?

While prefetching has the potential to improve the performance of the memory system, it is not a light-weight task and comes with its own overheads. Therefore, accuracy of what to prefetch is extremely important. Our first criterion in identifying the data to be prefetched is that the data should come from an *open row* that is in the row buffer. A prefetch from a row buffer can be fast and does not disturb the current contents of the row buffer. Within a row, there are many cache lines that we can potentially prefetch. We analyzed the line access patterns of different applications, specifically focusing on the order in which the lines in a row are accessed after the row is opened.

The three graphs in Figure 4.1 plot the line access patterns of some representative applications. Each application is run individually on a single core platform with one

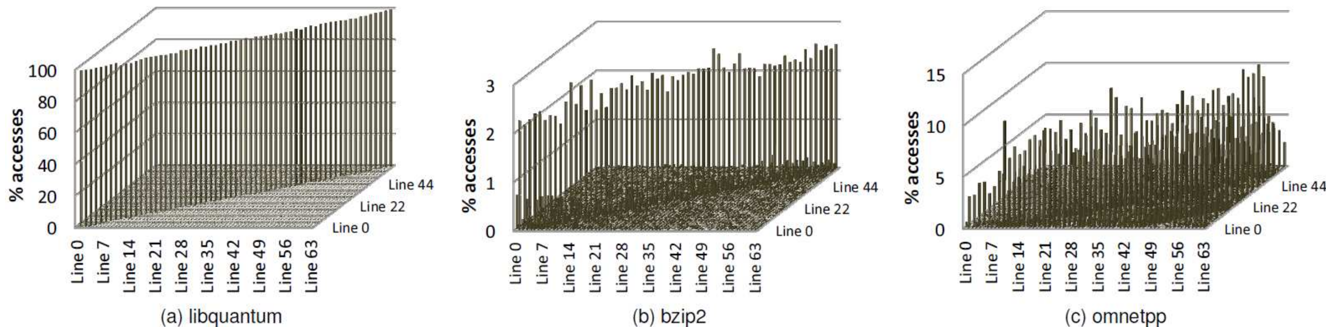


Fig. 4.1: Line access pattern graphs of some SPEC CPU2K6 applications.

memory controller to obtain these graphs. The xy-plane represents the line (each of size 64B) numbers within a row (of size 4KB), running from 0 to 63. For any point (x,y) in this plane, the value in the z-axis represents how frequently line y was accessed immediately after line x within the same row.

To quantify the line locality of an application, we define a metric called *next-line locality* (NLL), which is the average of all the values in the diagonal with offset 1 in the line access pattern graph (i.e., values at points $(i, i+1)$). This value represents the percentage of times a line is accessed immediately after its preceding line from the *same row*. Table 4.3 gives the NLL values of all the SPEC CPU2K6 applications. We see that the chances of accessing the next line in the same row after a line is accessed is about 36.8% over all applications. Extending the definition of next-line locality, *next-k-line locality* represents the percentage of times one of lines $(i+1, i+2, .., i+k)$ are accessed immediately after line i is accessed in the same row. The average next-k-line locality of our evaluated applications is around 43.7% for $k = 4$. This indicates that prefetching multiple subsequent lines may be a reasonable option in practice.

We selected next-line prediction for its simplicity to merely illustrate the benefits of memory-side prefetching and placing the data midway. Clearly, one could use a more sophisticated predictor like Adaptive Stream Detection Hur and Lin (2006). The predictor for the prefetching scheme is in itself orthogonal to the main contributions of this work. A sophisticated predictor has the potential to improve the accuracy of the memory prefetcher, further adding to the overall benefits one could achieve.

4.1.2 When to Prefetch?

It is of utmost importance to identify the "ideal time" to initiate a prefetch from the row buffer. The options are: (1) when a row buffer conflict happens, (2) when a row buffer hit happens, (3) when the row is first opened, or (4) when the memory bank and channel are idle.

Prefetch at Row Buffer Hit: Another opportunity for prefetching data from the row buffer is upon a row buffer hit (RBH). Along with a demand request, we issue prefetch requests for the next lines from the row buffer. We call this scheme *Prefetch at RBH*. By doing the prefetching at row buffer hit, we are not making the demand-request wait. Instead, we first service the demand-request and then perform our prefetch. Subsequent requests to those lines in that row (which were supposed to be row buffer conflicts or hits) can now become prefetch hits. One more advantage of *Prefetch at RBH* is the prefetches may more accurately reflect sequentiality.

A prefetch hit is much faster than a row buffer hit as it hides the latency of accessing the memory and is served directly from an on-chip buffer. Compared to Prefetch at RBC, the number of prefetches issued with Prefetch at RBH will be lower because of fewer row buffer hits compared to the row buffer conflicts in a system with multiple applications. Note that we prefetch only at *read* row buffer hits since we are prefetching to serve only read requests.

Prefetch Scheme	Critical Path	Locality	# of Prefetches
Prefetch at RBC	Yes	No	High
Prefetch at RBH	No	Yes	Low
Prefetch at Row ACT	No	No	High
Prefetch at Idle Times	No	Yes	High

Table 4.1: Characteristics of different prefetch schemes.

Table 4.1 gives a summary of the memory-side prefetching schemes discussed in this section. A good prefetching scheme should not be on the critical path, should consider locality before prefetching, and should not issue too many prefetches.

4.1.3 Where to Prefetch?

Once the questions of *what to prefetch* and *when to prefetch* are answered, the next question that comes up is *where to store the prefetched data?* Prefetched data should be stored on-chip to avoid off-chip latencies. Typically, core-side prefetchers bring the prefetched data to on-chip caches. In this work, we explore the option of storing the prefetched data in a separate buffer on-chip called *Prefetch Buffer* in the memory controller. As explained earlier, this cuts down a substantial part of the off-chip latency, while avoiding the problems of core-side prefetching.

The prefetch buffer is logically organized as a cache. Each row in this cache holds a prefetched memory block data and this data is identified uniquely by its corresponding memory address. The memory controller populates a new entry in the *prefetch buffer* when new data is prefetched from memory. Before issuing each memory read request, the memory controller does a lookup on the *prefetch buffer* and serves the request from the buffer if it is a hit.

We allocate a separate prefetch buffer at each memory controller. The prefetch buffer can be organized in two ways: (1) *shared prefetch buffer*, where any prefetch entry in the buffer can be used by any core, and (2) *private prefetch buffer*, where each core gets a specific set of prefetch entries in the buffer. A shared prefetch buffer suffers from the same problem of memory interference already observed by the memory controller queues. In this case, cores with high MPKI can take up more entries in the prefetch buffer, leading to fairness issues. Another problem with shared prefetch buffer is the extra cost of performing an associative search over a cache with relatively higher associativity. On the other hand, using private prefetch buffers reserves each core its own set of entries, and therefore, eliminates the interference between the cores at the prefetch buffer. While one can think that the data shared across applications or threads of a multithreaded application can lead to duplicate entries in the prefetch buffer which would not exist in a shared prefetch buffer, our experiments indicated that even with data sharing, the use of a shared prefetch buffer is not justified to offset the higher cost having a shared prefetch buffer. Therefore, we employ private per-core prefetch buffers.

4.1.4 Optimizations for Memory-Side Prefetching

Memory prefetching comes at the cost of extra pressure on the memory resources such as channels and banks. As a result, prefetching too frequently (e.g., along with every row buffer hit) can become an overkill. In some situations, serving a demand-request might be more beneficial than doing a prefetch (although prefetch will be done from an open row), because, while the prefetch is being done, all demand-requests in the bank queue get delayed. Further, application characteristics can vary dynamically throughout execution. There can be cases where an application or a phase of an application might not benefit from memory prefetching because of lack of locality. We implemented three optimizations to improve the effectiveness of the proposed memory-side prefetching.

• **Precharge on Prefetch:** After prefetching from an open row, there is a low chance of getting further immediate requests to that same row due to the filtering effect of the prefetch buffer. Consequently, we can precharge the row, thereby saving few cycles for subsequent requests to that bank.

• **Averting Costly Prefetches:** We do not issue prefetches when a high number of demand requests are queued, waiting for the channels or banks to become available. We calculate the total load on the memory bus and do prefetching only if the load is below a predetermined threshold (*Queueing Thresh*).

• **Prefetch Throttling:** We monitor the *usefulness* of prefetching for each application at runtime and reduce the prefetch degree dynamically for individual applications that do not benefit from it. For each application, the prefetch accuracy in the last epoch is calculated and based on its value relative to two predetermined thresholds *Accuracy Thresh1* and *Accuracy Thresh2* we select the prefetch degree for the next epoch. If the accuracy is greater than *Accuracy Thresh2* we select a high prefetch degree; if it is below *Accuracy Thresh1* we select a low prefetch degree; and if it is in between the two thresholds we select a moderate prefetch degree.

4.2 Experimental Evaluation

4.2.1 Setup

We evaluate our memory prefetching scheme using the Simics Magnusson et al. (2002) full-system simulator with GEMS Martin et al. (2005). We model a network-on-chip (NoC) based CMP with the MOESI_CMP_directory cache coherence protocol. We use Opal module of GEMS to simulate out-of-order cores. The cache and memory hierarchy are modeled using Ruby and the NoC is modeled using GARNET. Table 4.2 shows the important processor and memory parameters used in our experiments, and their default values. Later, we conduct a sensitivity analysis on some of these parameters.

Processor	32 cores at 2.4 GHz; ultra-sparc-iii-plus ISA out-of-order; 12-stage pipeline; issue width: 4
Network On Chip	8x4 2D mesh network; 2-stage pipelined router, 4VCs per port with 4-entry buffer each; 128b flit
Caches	64-byte cache line; 32KB L1D; 32KB L1I per core; 32MB banked shared LLC (S-NUCA) L1 hit latency: 3 cycles; L2 hit latency: 6 cycles
Memory	16GB; DDR3-1600; 4 memory channels; total 51.2 GBps bandwidth 1 DIMM, 2 ranks, and 16 banks at each channel
Memory Parameters	Row Buffer Hit: 42 cycles; Conflict: 102 cycles t_{CL}, t_{RP} and t_{RCD} = 10, 10, 10 memory cycles
Address Mapping	Page Interleaving at the Memory Controllers Cache Line Interleaving at the L2 caches
Memory Prefetch Parameters	Hit latency: 5 cycles; Max degree: 4 lines Timing: Prefetch at Row Buffer Hit 256KB, 128-entry, 32-way set associative prefetch buffer at each memory controller
Memory Prefetch Opt. Thresholds	Accuracy Thresh1: 10; Accuracy Thresh2: 25 Queueing Thresh: 48 requests
Cache Prefetch Parameters	Max Degree: 4; Prefetch distance: 24 Prefetch Window: 32 cache lines Train & Stream entries per core: 64

Table 4.2: Configuration of the evaluation platform.

4.2.2 Benchmarks

We created multiprogrammed workloads from applications in the SPEC CPU2K6 and SPEC CPU2K benchmark suites and multi-threaded workloads from SPEC OMP2K1 applications to evaluate our memory-side prefetching scheme. For each application, first we did a study on its memory characteristics. We specifically looked at the MPKI, *Row Buffer Locality* and *Next Line Locality* of each application when running alone on a single core platform. Table 4.3 shows the MPKI, row buffer hit rate (RBHR) and next line locality (NLL) of all the considered applications. We created three workload categories

Application	L2 MPKI	RBHR	NLL	Application	L2 MPKI	RBHR	NLL	Applicatio
470.lbm	34.54	36.49	5.88	183.equake	7.12	72.16	68.21	465.tonto
462.libquantum	31.66	97.03	98.90	436.cactus	5.83	13.57	10.62	444.namd
459.GemsFDTD	30.47	37.29	43.17	429.mcf	5.08	34.83	2.12	464.h264ref
179.art	25.57	80.89	6.01	435.gromacs	4.57	57.31	62.46	416.gamess
433.milc	21.47	66.99	37.55	171.swim	4.33	17.51	18.54	481.wrf
401.bzip2	20.63	3.10	2.47	434.zeusmp	2.88	49.59	49.12	458.sjeng
437.leslie3d	18.36	72.86	72.54	450.soplex	2.24	20.13	15.52	403.gcc
410.bwaves	17.10	66.09	67.45	454.calculix	1.20	93.19	21.02	453.povray
483.xalancbmk	16.77	76.30	14.31	473.astar	1.12	35.55	19.94	447.dealII
471.omnetpp	12.95	48.30	3.74	456.hmmer	0.84	38.69	34.32	400.perlbenc
482.sphinx3	11.99	29.50	26.25	445.gobmk	0.74	43.20	18.26	

Table 4.3: Memory characteristics of SPEC2006 applications.

(WL1 - High MPKI and High Locality, WL2 - High MPKI and Low Locality and WL3 - Low MPKI) with varying memory intensity and locality (NLL).

We created a workload category (WL4) with high locality applications which benefit core-side prefetching and a workload category (WL5) with mix of all types of applications. We created two additional workload categories (WL6 - 8 applications with 4 threads each and WL7 - one application with 32 threads) with multithreaded applications. In each workload category, we created four workloads of 32 applications each by randomly selecting applications from the corresponding category.

While running a workload, each of the 32 applications is bound to an individual core. The simulations are fast forwarded to 15 billion cycles, the caches are warmed up for 5 million instructions and the detailed simulation is run for 10 million instructions on the first core. We measure the overall performance of a workload using the *harmonic mean of the IPCs* of individual applications in the workload which represents both fairness and performance Luo et al. (2001). We analyze the effect of prefetching on other metrics like on-chip and off-chip latencies, L2 hit rates and row buffer hit rates. We further measure the following prefetching-specific metrics:

$$Prefetch\,Coverage_{memory-side} = \frac{Prefetch\,hits}{Total\,memory\,requests} \times 100,$$

$$Prefetch\,Coverage_{core-side} = \frac{Prefetch\,hits}{Prefetch\,hits + misses} \times 100,$$

$$Prefetch\,Accuracy = \frac{Prefetch\,hits}{Number\,of\,prefetched\,lines} \times 100$$

Prefetch coverage is the percentage of memory requests served from the prefetch buffer out of the total requests to memory controller. In the case of core-side prefetcher, *prefetch coverage* is the percentage of L2 misses avoided by prefetched lines. *Prefetch accuracy* is a measure of how many prefetches are actually useful.

4.2.3 Results and Analysis

To evaluate the benefits of our memory-side prefetching approach (*MSP*), we compare it against four different prefetching schemes: a sophisticated stream-based core-side prefetching scheme (*CSP*), a next-line (degree 1) core-side prefetching scheme, our memory-side prefetching scheme which pushes the prefetched data to the on-chip (L2) caches (*MSP-PUSH*), an existing memory side prefetching work Lin (2001) proposed for unicore systems, extended to multicores (*IDLE-PUSH*). The core-side prefetcher (*CSP*) we implemented is an *adaptive-stride stream-prefetcher* similar to the one in Liu et al. (2011) (and to the one adopted in current Intel Xeon Hegde (2008) and IBM Power Srinath et al. (2007) processors) that builds streams based on the L2 misses and prefetches data into the last-level (L2) cache (which is increased by the size of the total prefetch buffer (1MB) in this case). We added accuracy-based "dynamic throttling" Srinath et al. (2007) to the core-side prefetcher to improve its effectiveness. In addition, we implemented techniques presented in Ebrahimi et al. (2011) to reduce the effect of prefetch requests on demand requests at memory. We monitored the cache interference caused by our core-side prefetcher and identified that it is not a major problem in our configuration. We find that our core-side prefetcher has coverage and accuracy numbers similar to the prefetchers used in previous work Ebrahimi et al. (2011); Lee et al. (2008); Srinath et al. (2007). We also implemented a simple next-line prefetcher at the core although it is not expected to give better benefits than the sophisticated stream-based core-side prefetcher. Our memory-side prefetching (*MSP*) scheme prefetches at row buffer hits and stores the data in prefetch buffers at the memory controllers (on-chip). (*MSP-PUSH*) scheme is similar to (*MSP*), but does not use the prefetch buffers and instead pushes the prefetched data to the on-chip (L2) cache which is increased by 1MB. (*IDLE-PUSH*) is similar to Lin (2001) which uses the idle periods on the memory bus to prefetch data from open rows and pushes the prefetched data to on-chip (L2) caches. In the combined case (*CSP+MSP*), both the prefetchers work independently where the memory-side prefetcher brings data to prefetch buffers from DRAM and core-side prefetcher fetches data from either the prefetch buffer or memory to the on-chip (L2) caches.

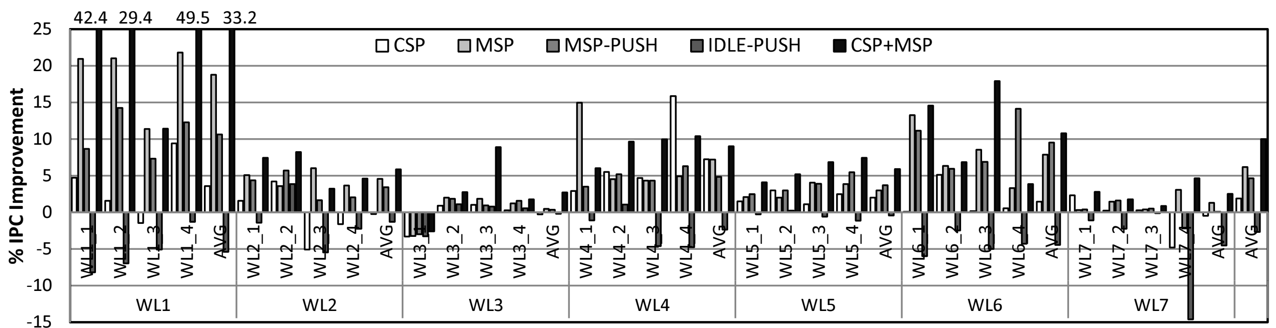


Fig. 4.2: Percentage IPC improvement over no prefetching with different prefetchers.

Figure 4.2 shows the percentage IPC improvements over the *no prefetching* case using five of the six schemes discussed above. The first bar for each workload shows the improvement with *CSP* scheme, the second, third, fourth and fifth bars show the improvements with the *MSP*, *MSP-PUSH*, *IDLE-PUSH*, and *CSP+MSP* schemes, respectively. The average IPC improvement over all 28 workloads (7 workload categories $\times$ 4 workloads/category) is 1.9% with the *CSP* scheme, -17.4% with next-line core-prefetcher (not shown in graph), 6.2% with *MSP*, 4.6% with *MSP-PUSH*, -3% with *IDLE-PUSH* and 10% with the *CSP+MSP* scheme. *CSP* scheme gave noticeable improvements (7.3%) only in workload category (WL4). Even by reserving a cache way just for prefetches the average IPC improvement with CSP increased to only 2.6%. We investigate the reason for this poor performance of *CSP* in the following paragraphs. In almost all workloads, memory-side prefetcher (*MSP*) performed better than the core-side prefetcher (*CSP*). The difference between (*CSP*) and (*MSP*) is more noticeable in high MPKI workloads (WL1, WL2 and WL6). The *MSP-PUSH* scheme suffers from the on-chip resource contention and loses some performance compared to *MSP*. The *IDLE-PUSH* scheme was not able to bring any performance improvement on average because of the following problems: (i) useful idle periods on the memory bus are rare (ii) the pushed data creates contention in the on-chip network and pollutes the caches. Accordingly, it performed well only in low MPKI workloads (WL3), and in all the other workloads its performance was worse than the baseline no prefetching case. The improvements obtained in the combined scheme (*CSP+MSP*) are higher than any other scheme in all the workloads. The improvements are especially good (average 33.2%) in high MPKI workloads (WL1). We want to emphasize that the improvement with *CSP+MSP* is larger than the sum of the improvements from *CSP* and *MSP* in many cases. This is because many core-side prefetches (in the combined case) hit in the prefetch buffer filled by the memory-side prefetcher in the *CSP+MSP* case which were originally accessing the DRAM and overwhelming the memory subsystem in the *CSP* case.

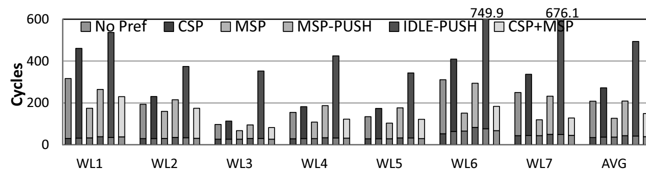


Fig. 4.3: Average on-chip and off-chip latencies for an LLC miss without prefetching and different prefetching schemes.

Figure 4.3 shows the average *on-chip* and *off-chip latencies* incurred by an L2 miss in different schemes (*No Pref*, *CSP*, *MSP*, *MSP-PUSH*, *IDLE-PUSH* and *CSP+MSP*). Six stacked bars are shown for each workload category. Each bar represents the average latencies in the four workloads in that category with a particular scheme. The first bar is the total miss latency without any prefetching, the second, third, fourth, fifth and sixth bars show the miss latencies with *CSP*, *MSP*, *MSP-PUSH*, *IDLE-PUSH* and *CSP+MSP* schemes respectively. Within each bar, the lower part shows the number

of cycles spent by a request on-chip, and the upper part shows the number of cycles spent in the memory controller. It can be seen that the core-side prefetching, even with sophisticated techniques like throttling and prefetch aware memory controllers, is still causing significant delays at the memory controller (35.8% increase on average). Our *MSP* scheme is very effective in reducing the off-chip latency (48.5% reduction on average) especially in high MPKI workloads (WL1, WL2, WL6 and WL7). *MSP-PUSH* and *IDLE-PUSH* schemes push all the prefetched data to caches and lead to higher on-chip latencies. They also show higher off-chip latencies because of the lack of prefetch buffer at memory controller in the case of *MSP-PUSH* and because of the interference of prefetch activity with demand requests in the *IDLE-PUSH* scheme. In the case of *CSP+MSP*, the increase in off-chip latency by the core-side prefetcher is balanced by the reduction in latency provided by the memory-side prefetcher.

Our memory-side prefetcher (*MSP*), being "DRAM state-aware" can also reduce the power consumption of DRAM by reducing the number of row activations and proactively retrieving data from open rows. In comparison to core-side prefetching, an inaccurate prefetch in our scheme (*MSP*) also wastes less energy as (i) it is done on an already-open row, and (ii) it does not use the on-chip network. The average DRAM power savings obtained with *MSP* (calculated using MICRON power calculator Micron (2009)) are 8.8% and 8.3% compared to no prefetching and *CSP* cases respectively.

From the above analysis it can be concluded that although core-side prefetcher (*CSP*) significantly improves L2 hit rates (18.9% on average) and does accurate prefetching (82.2% on average), it is not very effective in improving the overall performance of the system (average 1.9% in figure 4.2). This is due to the increase in queueing latency at the memory controllers (35.8% on average as shown in figure 4.3) because of the bursty nature of core-side prefetch requests. To confirm this fact, we ran the same simulations with a constant memory latency (200 cycles) and the core-side prefetcher gave an average improvement of 10.4%. In fact, we could not find any work in recent literature which shows significant performance improvements with core-side prefetching on a many(32)-core system when full memory subsystem and on-chip network is modeled. We believe, this problem of off-chip queueing becomes more prominent as we move to larger core counts and core-side prefetching becomes far less effective. On the other hand, our memory-side prefetcher is a low overhead solution which can scale with the prefetching requirements of larger number of cores. However, pushing the prefetched data to on-chip caches is also not a good idea because of on-chip network and cache contention problems. Finally, we conclude that, combined prefetching (*CSP+MSP*) leverages the advantages of both the prefetching schemes and yields better benefits than either of them.

Chapter 5

Mobile Memory

5.1 Problem Statement

DRAM stalls are high in current SoCs and this will only worsen as IPs performance scale. Typically, DRAM is shared between the cores and IPs and is used to transfer data between them. There is a high degree of data movement and this often results in a high contention for memory controller bandwidth between the different IPs Jog et al. (2013). Figure 5.1 shows the *memory bandwidth obtained* by two of our applications: `YouTube` and `Skype` with a 6.4 GBPS memory. One can notice the *burstiness* of the accesses in these plots. Depending on the type of IPs involved, frames get written to memory or read from memory at a certain rate. For example, cameras today can capture video frames of resolution 1920x1080 at 60 FPS and the display refreshes the screen with these frames at the same rate (60 FPS). Therefore, 60 bursts of memory requests from both IPs happen in a second, with each burst requesting one complete frame. While the request rate is small, the data size per request is high – 6MB for a 1920x1080 resolution frame (this will increase with 4K resolutions Engwell (2013)). If this amount of bandwidth cannot be catered to by the DRAM, the memory controller and DRAM queues fill up rapidly and in turn the devices and accelerators start experiencing performance drops. The performance drop also affects battery life as execution time increases. In the right side graph in Figure 5.1, whenever the camera (CAM) initiates its burst of requests, the peak memory bandwidth consumption can be seen (about 6 GBPS). We also noticed that the average memory latency more than doubles in those periods, and memory queues sustain over 95% utilization.

To explain how much impact the memory subsystem and the system-agent can have on IPs' execution time (*active cycles* during which the IPs remains in active state), in Figure 5.2, we plot the total number of cycles spent by an IP in processing data and in data stalls. Here, we use "data stall" to mean the number of cycles an IP stalls for data without doing any useful computation, after issuing a request to the memory. We observe from Figure 5.2 that the video decoder and video encoder IPs spend most of their time processing the data, and do not stress the memory subsystem. IPs that have very small compute time, like the audio decoder and sound engine, demand very high bandwidth than what memory can provide, and thus tend to stall more than compute. Camera IP and graphics IP, on the other hand, send bursts of requests for large frames of data at regular intervals. Here as well, if memory is not able to meet the high bandwidth or has high latency, the IP remains in the high-power mode stalling for the requests. The high data stalls seen in Figure 5.2 translate to frame drops which is shown in Figure 5.3 (for 5.3 GBPS memory bandwidth). We see that on average 24% of the frames are dropped with the default baseline system, which can hurt user experience with the device. With

higher memory bandwidths (2x and 4x of the baseline bandwidth), though the frame drops decrease, they still do not improve as much as the increase in bandwidth. Even with 4x baseline bandwidth, we observe more than 10% frame drops (because of higher memory latencies).

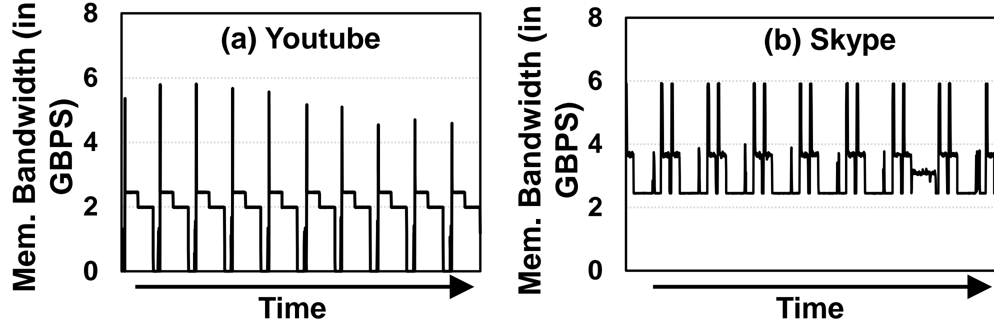


Fig. 5.1: Bandwidth usage of Youtube and Skype over time.

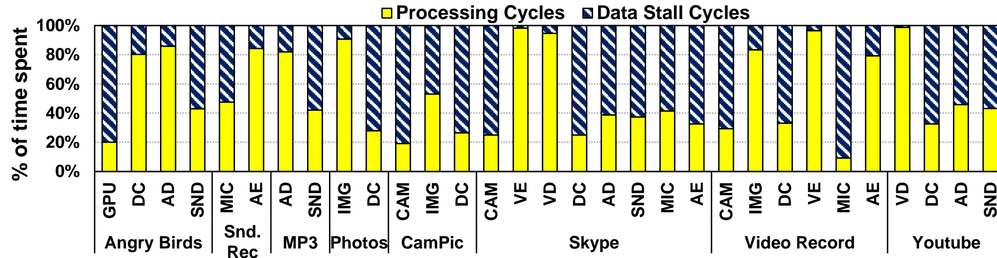


Fig. 5.2: Total data stalls and processing time in IPs during execution.

As user demands increase and more use-cases need to be supported, the number of IPs in the SoC is likely to increase Steve Scheirey (2013) along with data sizes Engwell (2013). Even as the DRAM speeds increase, the need to go off-chip for data accesses places a significant bottleneck. This affects performance, power and eventually the overall user experience. To understand the severity of this problem, we conduct a simple experiment shown in Figure 5.4, demonstrating how the cycles per frame vary across the base system (given in Table 5.1) and when the IPs compute at half their base speed (last generation IPs), and twice their speed (next generation) etc. For DRAM, we varied the memory throughput by varying the LPDDR configurations. We observe from the results in Figure 5.4 that the percentage of data stalls increases as we go from one generation to the next. Increasing the DRAM peak bandwidth alone is *not* sufficient to match the IP scaling. We require solutions that can tackle this problem within the SoC.

Further, to establish the maximum gains that can be obtained if we had an "ideal and perfect memory", we did a hypothetical study of *perfect memory with 1 cycle latency*. The cycles-per-frame results with this perfect memory system are shown in Figure 5.5.

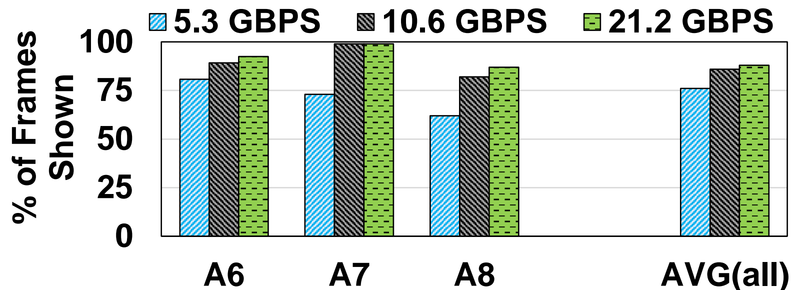


Fig. 5.3: Percentage of frames completed in a subset of applications with varying memory bandwidths.

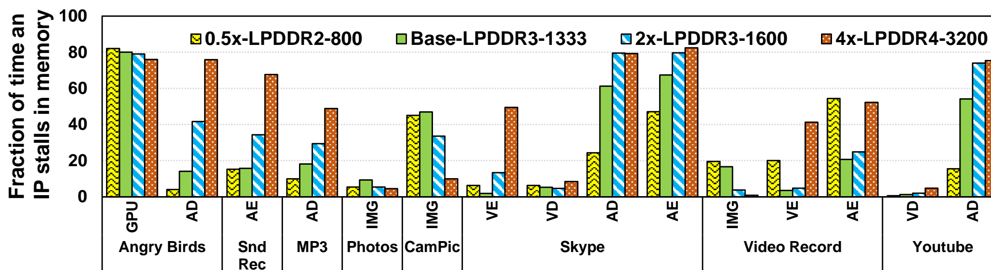


Fig. 5.4: Trends showing increase of percentage of data stalls with each newer generation of IPs and DRAMs.

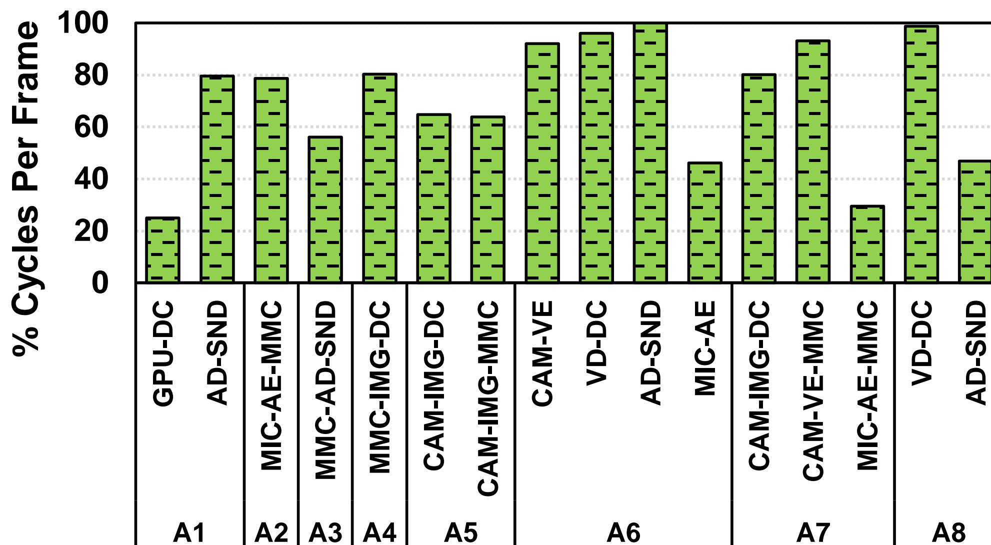


Fig. 5.5: Percentage reduction in Cycles-Per-Frame in different flows with a perfect memory configuration.

As expected, we observed drastic reduction in cycles per frames across applications and IPs (as high as 75%). In some IPs, memory is not a bottleneck and those did not show improved benefits. From this data, we conclude that reducing the memory access times does bring the cycles per frame down, which in turn boosts the overall application performance. Note that, this perfect memory does not allow any frames to be dropped.

5.2 Evaluation Platform

Handheld/mobile platforms commonly run applications that rely on user inputs and are interactive in nature. Studying such a system is tricky due to the non-determinism associated with it. To enable that, we use GemDroid Chidambaram Nachiappan et al. (2014), which utilizes Google Android's open-source emulator Google (2013) to capture the complete system-level activity. This provides a complete memory trace (with cycles between memory accesses) along with all IP calls when they were invoked by the application. We extended the platform by including DRAMSim2 Rosenfeld et al. (2011) for accurate memory performance evaluation. Further, we enhanced the tool to extensively model the system agent, accelerators and devices in detail.

Component	Configuration
Processor	ARM ISA; 4-core processor; Clocked at 2 GHz; OoO w/issue width: 4
Caches	32 KB L1-I; 32KB L1-D; 512 KB L2
Memory	Till 2 GB reserved for cores. 2GB to 3GB reserved for IPs. LPDDR3-1333; 1 channel; 1 rank; 8 Banks 5.3 GBPS peak bandwidth; t_{CL},t_{RP},t_{RCD} = 12, 12, 12 ns
System Agent	Frequency: 500 MHz; Interconnect latency: 1 cycle per 16 Bytes Memory Transaction-Q.: 64 entries; Bank-Q.: 8 entries
IPs and System Parameters	All IPs run at 500Mhz frequency Aud.Frame: 16KB frame; Vid.Frame: 4K (3840x2160) Camera Frame: 1080p (1920x1080) Input Buffer Sizes: 16-32KB; Output Buffer Sizes: 32-64KB Enc/Decoding Ratio: VD→1:32; AD→1:8;

Table 5.1: Platform configuration.

Specifications of select IPs frequency, frame sizes and processing latency are available from SoC (2011). For completeness, we give all core parameters, DRAM parameters, and IP details in Table 5.1. Note that the techniques discussed in this work are generic and not tied to specific microarchitectural parameters.

While most of the applications that we use in this work are well known, we would like to mention a few details. In our setup, all video-based applications like `youtube`, `angry birds game`, and `photo-gallery` work with 1080x1920 HD frames simulating at least 200 frames in total at 60 FPS (frames-per-second). `Audio-playback` plays a

IP Abbr.	Expansion	IP Abbr.	Expansion
VD	Video Decoder	AD	Audio Decoder
DC	Display Controller	VE	Video Encoder
MMC	Flash Controller	MIC	Microphone
AE	Audio Encoder	CAM	Camera
IMG	Imaging	SND	Sound

Table 5.2: Expansions for IP abbreviations.

Id	Application	IP Flows
A1	Angry Birds	AD - SND; GPU - DC
A2	Sound Record	MIC - AE - MMC
A3	Audio Playback (MP3)	MMC - AD - SND
A4	Photos Gallery	MMC - IMG - DC
A5	Photo Capture (Cam Pic)	CAM - IMG - DC; CAM - IMG - MMC
A6	Skype	CAM - VE; VD - DC; AD - SND; MIC - AE
A7	Video Record	CAM - VE - MMC; MIC - AE - MMC
A8	Youtube	VD - DC; AD - SND

Table 5.3: IP flows in our applications.

320Kbps music file, while audio-record/encoding occurs at 44KHz. Input from CAM is set up with a 4K frame for video recording (again at 60 FPS). `Youtube` downloads data over the network before video-decoding. The specifications used are derived from current systems in the marketSamsung (2014); Apple.

5.3 IP-to-IP Data Reuse

This section explores performance optimization opportunities that exist in current designs and whether existing solutions can exploit that.

5.3.1 Data Reuse and Reuse Distance

In a flow, data get read, processed (by IPs) and written back. The producer and consumer of the data could be two different IPs or sometimes even the same IP. We capture this IP-to-IP reuse in Figure 5.6, where we plotted the physical addresses accessed by the core and other IPs for `YouTube` application. Note that this figure only captures a very small slice of the entire application run. Here, we can see that the display-controller (DC) (red points) reads a captured frame from a memory region that was previously written to by video decoder (black points). Similarly, we can also see that the sound-engine reads from an address region where audio-decoder writes. This clearly shows that the data gets reused repeatedly across IPs, but the reuse distances can be very high.

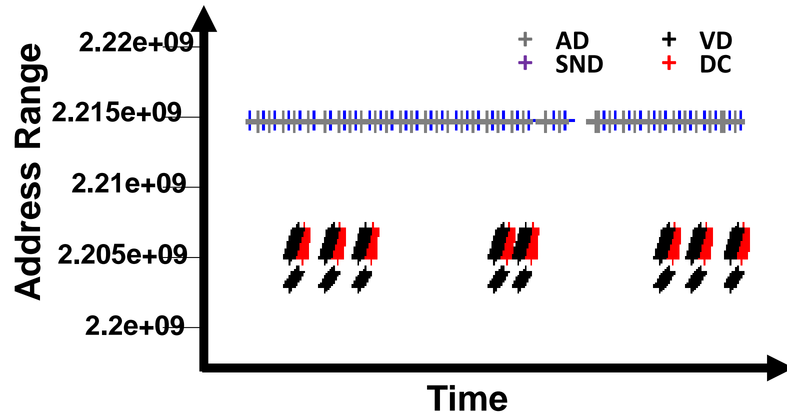


Fig. 5.6: Data access pattern of IPs in YouTube application.

As mentioned in Section 2.1.1, when a particular application is run, the same physical memory regions get used (over time) by an IP for writing different frames. In our current context, the reuse we mention is only between the producer and consumer IPs for a particular frame and nothing to to do with frames being rewritten to the same addresses. Due to frame rate requirements, reuse distances between display frame based IPs were more than tens of milli-seconds, while audio frame based IPs were less than a milli-second. Thus, there is a large variation across producer-consumer reuse distances across IPs that process large (display) frames (e.g., VD, CAM) and IPs that process smaller (audio) frames (e.g., AD, AE).

5.3.2 Converting Data Reuse into Locality

Given the data reuse, the simplest solution is to place a on-chip cache and allow the multiple IPs to share it. The expectancy is that caches are best for locality and hence they should work. In this subsection, we evaluate the impact of adding such a shared cache to hold the data frames. Typical to conventional caches, on a cache-miss, the request is sent to the transaction queue. The shared cache is implemented as a direct-mapped structure, with multiple read and write ports, and multiple banks (with a bank size of 4MB), and the read/write/lookup latencies are modeled using CACTI Shivakumar and Jouppi (2001). We evaluated multiple cache sizes, ranging from 4MB to 32MB, and analyzed their hit rates and the reduction in cycles taken per frame to be displayed. We present the results for 4MB, 8MB, 16MB and 32MB shared caches in Figure 5.7 and Figure 5.8 for clarity. They capture the overall trend observed in our experiments. In our first experiment, we notice that as the cache sizes increase, the cache hit rates either increase or remain the same. For applications like `Audio Record` and `Audio Play` (with small frames), we notice 100% cache hit rates from 4MB cache. For other applications like `Angry Birds` or `Video-play` (with larger frames), a smaller cache does not suffice. Thus, as we increase the cache capacity, we achieve higher hit rates. Interestingly, some applications have very low cache hit rates even with large caches. This can be attributed to two main reasons. First, frame sizes are very large to fit even two frames in a large 32MB cache (as in the case of `YouTube` and `Gallery`). Second, and most importantly,

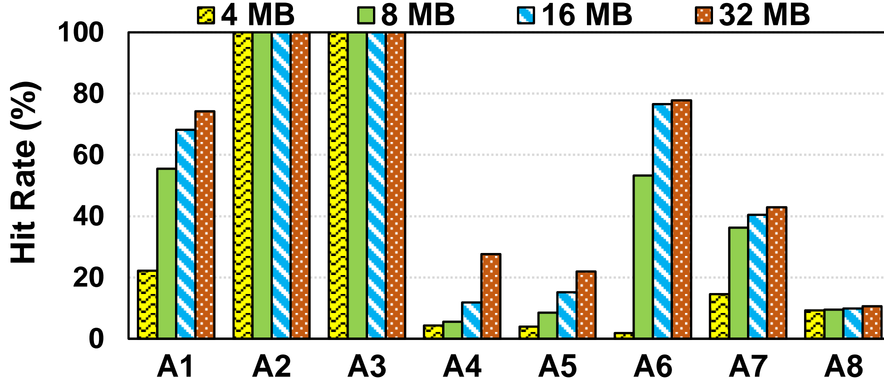


Fig. 5.7: Hit rates under various cache capacities.

if the reuse distances are large, data gets kicked out of caches by the other flows in the system or by other frames in the same flow. Applications with large reuse distances like `Video-record` exhibit such behavior.

In our second experiment, we quantify the performance benefits of having such large shared caches between IPs, and give the average cycles consumed by an IP to process a full-frame (audio/video/camera frame). As can be seen from Figure 5.8, increasing the cache sizes does not always help and there is no optimal size. For IPs like SND and AD, the frame sizes are small and hence a smaller cache suffices. From there on, increasing cache size increases lookup latencies, and affects the access times. In other cases, like DC, as the frame sizes are large, we observe fewer cycles per frame as we increase the cache size. For other accelerators with latency tolerance, once their data fits in the cache, they encounter no performance impact.

Further, scaling cache sizes above 4MB is not reasonable due to their area and power overheads. Figure 5.9 plots the overheads for different cache sizes. Typically, handhelds operate in the range of 2W – 10W, which includes everything on the device (SoC+display+network). Even the 2W consumed by the 4MB cache will impact battery life severely.

Summary: To summarize, the high number of memory stalls is the main reason for frame drops, and large IP-to-IP reuse distances is the main cause for large memory stalls. Even large caches are not sufficient to capture the data reuse and hence, accelerators and devices still have considerable memory stalls. All of these observations led us to re-architect how data gets exchanged between different IPs, paving way for better performance.

5.4 Sub-Framing

Our primary goal is to reduce the IP-to-IP data reuse distances, and thereby reduce data stalls, which we saw were major impediment to performance in Section 5.1. To achieve this, we propose a novel approach of *sub-framing* the data. One of the commonly used compiler techniques to reduce the data reuse distance in loop nests that manipulate array data is to employ *loop tiling* Song and Li (1999); Lim and Lam (1997).

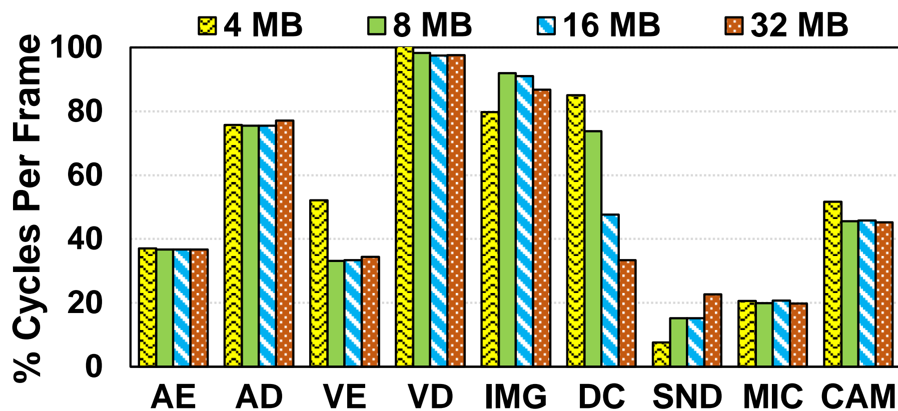


Fig. 5.8: Cycles Per Frame under various cache capacities.

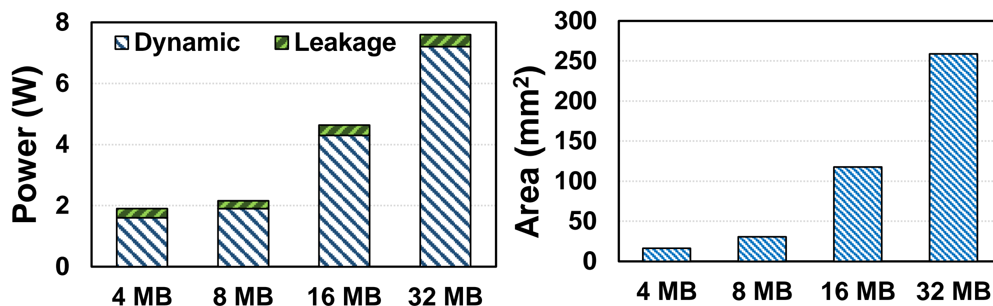


Fig. 5.9: Area and power-overhead with large shared caches.

It is the process of partitioning a loop's iteration space into smaller blocks (tiles) in a manner that the data used by the loop remains in the cache enabling quicker reuse. Inspired by tiling, we propose to break the data frames into smaller *sub-frames*, that reduces IP-to-IP data reuse distances.

In current systems, IPs receive a request to process a data frame (it could be a video frame, audio frame, display frame or image frame). Once it completes its processing, the next IP in the pipeline is triggered, which in-turn triggers the following IP once it completes its processing and so on. In our solution, we propose to sub-divide these data frames into smaller sub-frames, so that once IP1 finishes it's first subframe, IP2 is invoked to process it. In the following sections, we show that this design reduces the hardware requirements to store and move the data considerably thereby bringing both performance and power gains. The granularity of the subframe can have a profound impact on various metrics.

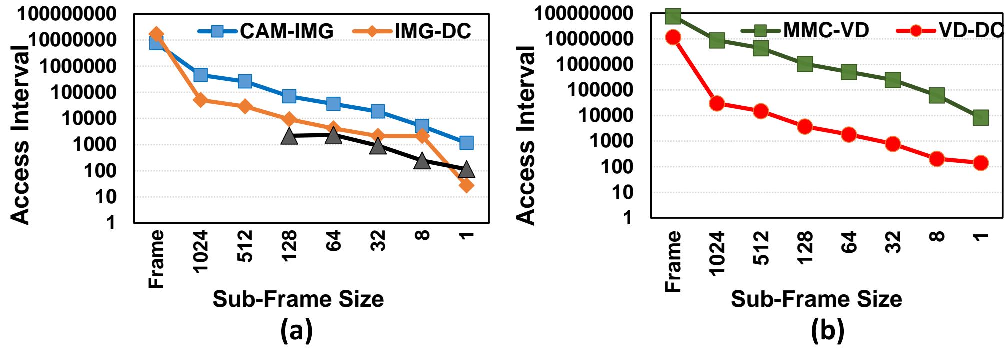


Fig. 5.10: IP-to-IP reuse distance variation with different sub-frame sizes. Note that the y-axis is in the log scale.

To quantify the effects of subdividing a frame, we varied the sub-frame sizes from 1 cache line to the current data frame size, and analyzed the reuse distances. Figure 5.10 plots the reduction in the IP-to-IP reuse distances (on y-axis, plotted on *log-scale*), as we reduced the size of a sub-frame. We can see from this plot an inverse exponential decrease in reuse distances. In fact, for very small sub-frame sizes, we see reuse distances in less than 100 cycles. To capitalize on such small reuse distances, we explore two techniques – *flow-buffering* and opportunistic IP-to-IP *request short-circuiting*.

5.4.1 Flow-Buffering

In Section 5.3.2, we showed that even large caches were not very effective in avoiding misses. This is primarily due to very large reuse distances that are present between the data-frame write by a producer and the data-frame read by a consumer. With sub-frames, the reuse distances reduce dramatically. Motivated by this, we now re-explore the option of caching data. Interestingly, in this scenario, caches of much smaller size can be far more effective (low misses). The reuse distances resulting from sub-framing are so small that even having a structure with few cache-lines is sufficient to

capture the temporal locality offered by IP pipelining in SoCs. We call these structures as *flow-buffers*. Unlike a shared cache, the flow-buffers are private between any two IPs. This design avoids the conflict misses seen in a shared cache (fully associative has high power implications). These flow-buffers are write-through. As the sub-frame gets written, the sub-frame is written to memory. The reason for this design choice is discussed next.

In a typical use-case involving data flow from IP-A→IP-B→IP-C, IP-A gets its data from the main-memory and starts computing it. During this process, as it completes a sub-frame, it writes back this chunk of data into the flow-buffer between IP-A and IP-B. IP-B starts processing this sub-frame from the flow-buffer (in parallel with IP-A working on another sub-frame) and writes it back to the flow-buffer between itself and IP-C. Once IP-C is done, the data is written into the memory or the display. Originally, every read and write in the above scenario would have been scheduled to reach the main memory. Now, with the flow-buffers in place, all the requests can be serviced from these small low-latency cost and area efficient buffers.

Note that, in these use-cases, cores typically run device driver code and handle interrupts. They have minimal data frames processing. Consequently, we do not incorporate flow-buffers between core and any other accelerator. Also, when a use-case is in its steady-state (for example, a minute into running a video), the IPs are in the active state and quickly consume data. However, if an IP is finishing up on an activity or busy with another activity or waking up from sleep state, the sub-frames can be overwritten in the flow-buffer. In that case, based on sub-frame addresses, the consumer IP can find its data in the main memory since the flow-buffer is a write-through buffer. In our experiments, discussed later in Section 5.6, we found that a flow-buffer size of 32 KB provides a good trade-off between avoiding a large flow-buffer and sub-frames getting overwritten.

5.4.2 IP-IP Short-circuiting

The flow-buffer solution requires an extra piece of hardware to work. To avoid the cost of adding the flow-buffers, an alternate technique would be to enable consumers directly use the data that their producers provide. Towards that, we analyzed the average round-trip delays of all accesses issued by the cores or IPs (shown in Figure 5.11) and found requests spend maximum time queuing in the memory subsystem. `MC Trans Queue` shows the time taken from the request leaving the IP till it gets to the head of the transaction queue. The next part `MC Bank Queue`, is the time spent in bank queues. This is primarily determined by whether the data access was a row buffer hit, or miss. And, finally `DRAM` shows the time for DRAM accessing along with the response back to the IPs.

As can be seen, most of the time is spent in the memory transaction queues (~100 cycles). This means that data that could otherwise be reused lies idle in the memory queues and we use this observation towards building an opportunistic IP-to-IP short-circuiting technique, similar in concept to "store-load forwarding" in CPU coresSha

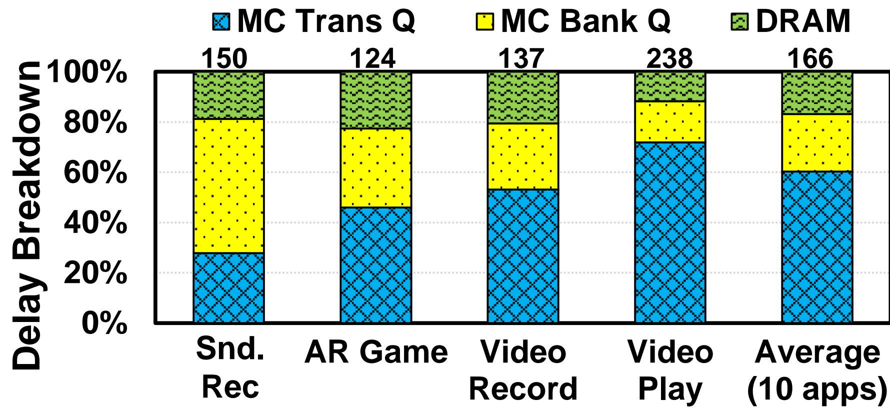


Fig. 5.11: Delay breakdown of a memory request issued by IPs or cores. The numbers above the bar give the absolute cycles.

et al. (2005); Loh et al. (2002)[1] though our technique is in between different IPs. There are correctness and implementation differences, which we highlighted in the following paragraphs/sections.
IPs usually load the data frames produced by other IPs. Similar to store-load forwarding, if the consumer IP's load requests can be satisfied from the memory transaction queue or bank queues, the memory stall time can be considerably reduced. As the sub-frame size gets smaller, the probability of a load hitting a store gets higher. Unlike the flow-buffers discussed in Section 5.4.1, store data does not remain in the queues till they are over-written. This technique is opportunistic and as the memory bank clears up its entries, the request moves from the transaction queue into the bank queues and eventually into main memory. Thus, the loads need to follow the stores quickly, else it has to go to memory. This distance between the consumer IP load request and producer IP store request depends on how full the transaction and bank queues are. In the extreme case, if both the queues (transaction-queue and bank-queue) are full, the number of requests that a load can come after a store will be the sum of the number of entries in the queues.

The overhead of implementing the IP-IP short-circuiting is not significant since we are using pre-existing queues present in the system agent. The transaction and bank queues already implement an associative search to re-order requests based on their QoS requirements and row-buffer hits, respectively Rixner et al. (2000). Address-searches for satisfying core loads already exist and these can be reused for other IPs. As we will show later, this technique works only when the sub-frame reuse distance is small.

[1] Core requests spend relatively insignificant amount of time in transaction queues as they are not bursty in nature. Due to their strict QoS deadlines, they are prioritized over other IP requests. They spend more time in bank queues and in DRAM.

5.4.3 Effects of Sub-framing Data with Flow-Buffering and IP-IP Short-circuiting

The benefits of sub-framing are quantified in Figure 5.12 in terms of hit rates when using flow-buffering and IP-IP short-circuiting. We can see that the buffer hit rates increase as we increase the size of flow-buffers, and saturate when the size of buffers are in the ranges of 16KB to 32KB. The other advantage of having sub-frames is the reduced bandwidth consumption due to the reduced number of memory accesses. As discussed before, accelerators primarily face bandwidth issues with the current memory subsystem. Sub-framing alleviates such bottleneck by avoiding fetching every piece of data from memory. Redundant writes and reads to same addresses are avoided. Latency benefits of our techniques, as well as their impact on user experience will be given later in Section 5.6.

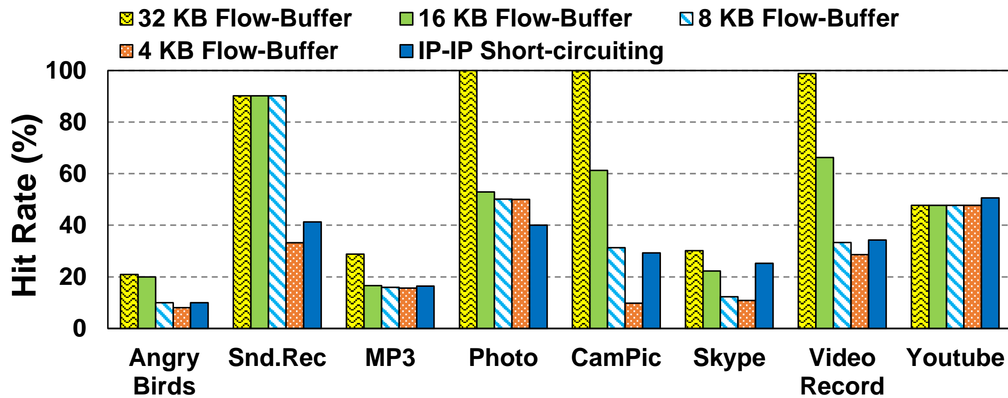


Fig. 5.12: Hit rates with flow-buffering and IP-IP short-circuiting.

5.5 Implementation Details

In implementing our sub-frame idea, we account for the probable presence of dependencies and correctness issues resulting from splitting frames. Below, we discuss the correctness issue and the associated intricacies that need to be addressed to implement sub-frames. We then discuss the software, hardware and system-level support needed for such implementations.

5.5.1 Correctness

We broadly categorize data frames into the following types – (i) video, (ii) audio, (iii) graphics display, and (iv) the network packets. Of these, the first three types of frames are the ones that usually demand sustained high bandwidth with the frame sizes varying from a megabyte to tens of MBs. In this work, we address only the first three types of frames, and leave out network packets as the latency of network packet transmission is considerably higher compared to the time spent in the SoC.

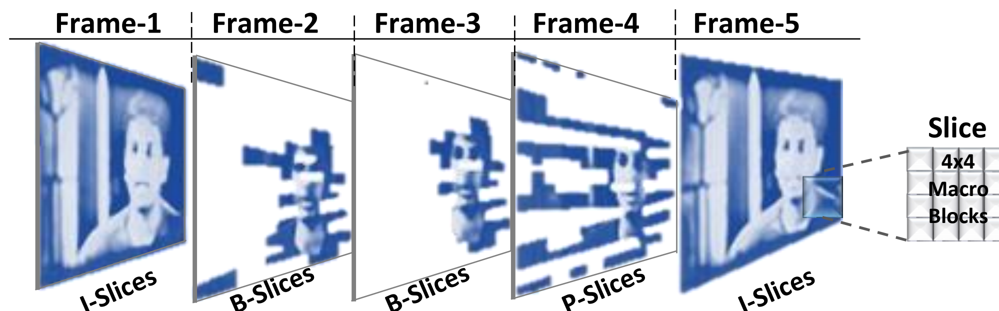


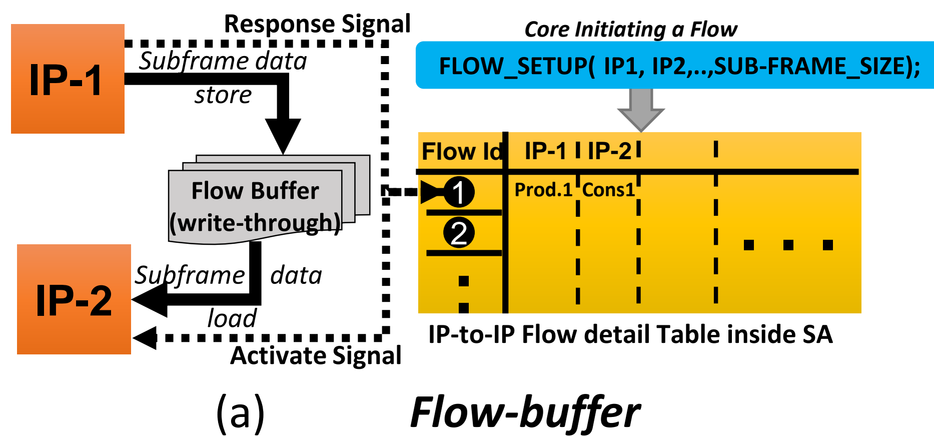
Fig. 5.13: Pictorial representation showing the structure of five consecutive video frames.

Video and Audio Frames Encoding and decoding, abbreviated as *codec* is compression and decompression of data that can be performed at either hardware or software layer. Current generation of smartphones such as Samsung S5 Samsung (2014) and Apple iPhone Apple have multiple types of codes embedded in their phone.

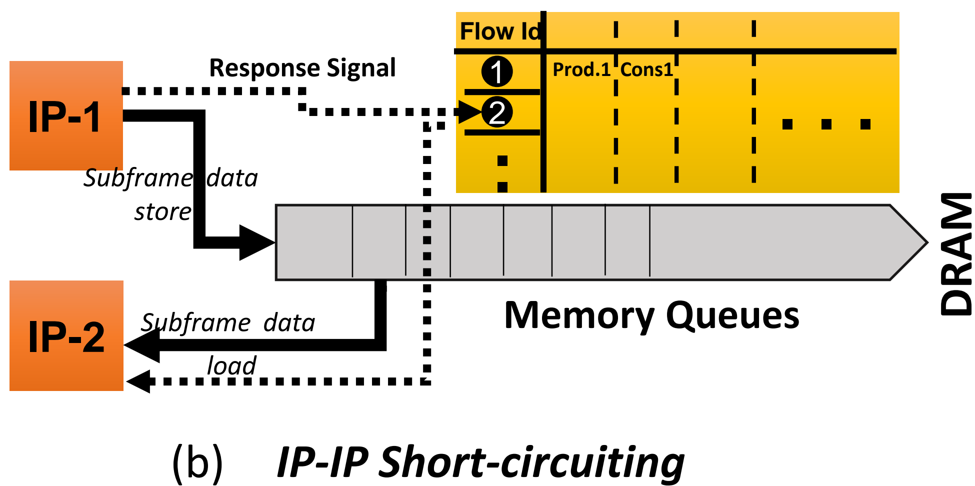
Video Codecs: First, let us consider the flows containing video frames, and analyze the correctness of sub-dividing such large frames into smaller ones. Among the video codecs, the most commonly used are H.264 (MPEG-4) or H.265 (MPEG-H, HEVC) codecs. Let us take a small set of video frames and analyze the decoding process. The encoding process is almost equivalent to the inversion of each stage of decoding. As a result, similar principles apply there as well. Figure 5.13 shows a video clip in its entirety, with each frame component named. A high-quality HD video is made up of multiple frames of data. Assuming a default of 60 FPS, the amount of data needed to show the clip for a minute would be 1920x1080(screen resolution) x 3(bytes/pixel) x 60 (frame rate) x 60 (seconds) = 21.35 GB. Even if each frame is compressed individually and stored on today's hand-held devices, the amount of storage available would not permit it. To overcome this limitation, codecs take advantage of the temporal redundancy present in video frames, as the next frame is usually not very different from the previous frame.

Each frame can be dissected into multiple slices. Slices are further split into macroblocks, which are usually a block of 16x16 pixels. These macroblocks can be further divided into finer granularities such as sub-macroblocks or pixel blocks. But, we do not need such fine granularities for our purpose. Slices can be classified into 3 major types: I-Slice (independent or intra slices), P-Slice (predictive), and B-Slice (bi-directional) Schwarz et al. (2007) as depicted in Figure 5.13. [2] I-slices have all data contained in them, and do not need motion prediction. P-slices use motion prediction from one slice which belongs to the past or future. B-slices use two slices from past or the future. Each slice is an independent entity and can be decoded without the need for any other slice in the *current frame*. P- and B-slices need slices from a previous or next frame only.

[2]Earlier codecs had frame level classification instead of slice level. In such situations, I-frame is constructed as a frame with only I-slices.



(a) *Flow-buffer*



(b) *IP-IP Short-circuiting*

Fig. 5.14: High level view of the SA that handles sub-frames.

In our sub-frame implementation, we choose *slice-level granularity* as the finest level of sub-division to ensure correctness *without having any extra overhead of synchronization.* As slices are independently decoded in a frame, the need for another slice in the frame does not arise, and we can be sure that correctness is maintained. Sub-dividing any further would bring in dependencies, stale data and overwrites.

Audio Codecs: Audio data is coded in a much simpler fashion than video data. An audio file has a large number of frames, with each audio frame having the same number of bits. Each frame is independent of another and it consist of a header block and data block. Header block (in MP3 format) stores 32-bits of metadata about the coming data block frame. Thus, each audio frame can be decoded independently of another as all required data for decoding is present in the current frames header. Therefore, using a full audio frame as a sub-frame would not cause any correctness issue.

Graphics Rendering: Graphics IPs already employ tiled rendering when operating on frames and for the display rendering. These tiles are similar to the sub-frames proposed in this work. A typical tiled rendering algorithm first transforms the geometries to be drawn (multiple objects on the screen) into screen space and assigns them into tiles. Each screen-space tile holds a list of geometries that needs to be drawn in that tile. This list is sorted front to back, and the geometry behind another is clipped away and only the ones in the front are drawn to the screen. GPUs renders each tile separately to a local on-chip buffer, which is finally written back to main-memory inside a framebuffer region. From this region, the display controller reads the frame to be displayed to be shown on screen. All tiles are independent of each other, and thus form a sub-frame in our system.

5.5.2 OS and Hardware Support

In current systems, IPs are invoked sequentially one-after-another per frame. Let us revisit the example considered previously – a flow with 3 IPs. The OS, through device drivers, calls the first IP in the flow. It waits for the processing to complete and the data to be written back to memory and then calls the second IP. After the second IP finishes its processing, the third IP is called. With sub-frames, when the data is ready in the first IP, the second IP is notified of the incoming request so that it can be ready (by entering to the active state from a low power state) when the data arrives. We envision that the OS can capture this information through a library (or multiple libraries) since the IP-flows for each application are pretty standard. In Android Google for instance, there is a layer of libraries (Hardware Abstraction Layer – HAL) that interface with the underlying device drivers and these HALs are specific to IPs. As devices evolve, HAL and the corresponding drivers are expected to enable access to devices to run different applications. By adding an SA HAL and its driver counterpart to communicate the flow information, we can accomplish our requirements. From the application's perspective, this is transparent since the access to the SA HAL happens from within other HALs as they are requested by the applications. Figure 5.14 shows a high level view of the sub-frame implementation in SA along with our short-circuiting techniques. From a hardware perspective, to enable sub-framing of data, the SA needs to have a small matrix of all IPs – rows corresponding to producers and columns to consumers. Each entry in the

row is 1 bit per IP. Currently, we are looking at about 8 IPs, and this is about 8 bytes in total. In future, even as we grow to 100 IPs, the size of the matrix is small. As each IP completes its sub-frame, the SA looks at its matrix and informs the consumer IP. In situations where we have multiple flows (currently Android allows two applications to run simultaneously Report (2012)) with an IP in common, the entries in the SA for the common IP can be swapped in or out along with the context of the application running. This will maintain the correct consumer IP status in the matrix for any IP.

5.6 Evaluation

In this section, we present the performance and power benefits obtained by using sub-frames compared to the conventional baseline system which uses full frames in IP flows. We used a modified version of the GemDroid infrastructure Chidambaram Nachiappan et al. (2014) for the evaluation. For each application evaluated, we captured and ran the traces either to completion or for a fixed time. The trace lengths varied from about 2 secs to 30 secs. Usually this length is limited by frame sizes we needed to capture. Workloads evaluated are listed in Table 5.3 and the configuration of the system used is described in Section 5.2 and Table 5.1. For this evaluation we used a sub-frame size of 32 cache lines (2KB). The transaction queue and bank queues in SA can hold 64 entries (totaling 8KB). For the flow buffer solution, we used a 32 KB buffer (based on the hit rates observed in Figure 5.12).

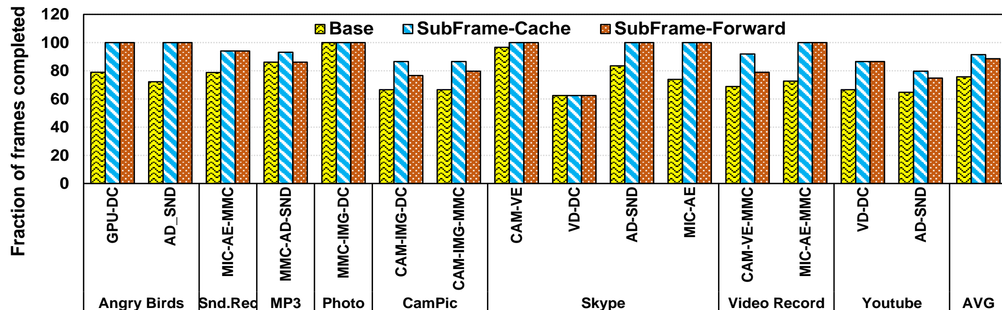


Fig. 5.15: Percentage of Frames Completed (Higher the better).

User Experience: As a measure of user experience, we track the number of frames that could complete in each flow. The more frames that get completed, lesser the frame drops and better is the user experience. Figure 5.15 shows the number of frames completed in different schemes. The y-axis shows the percentage of frames completed out of the total frames in an application trace. The first bar shows the frames completed in the baseline system with full frame flows. The second and third bars show the percentage of frames completed with our two techniques. In baseline system, only around 76% of frames were displayed. By using our two techniques, the percentage of frames completed improved to 92% and 88%, respectively. Improvements in our schemes are mainly attributed to the reduced memory bandwidth demand and improved memory

latency as the consumer IP's requests are served through the flow-buffers or by short-circuiting the memory requests. The hit-rates of consumer's requests were previously shown in Figure 5.12. In some cases, flow-buffers perform better than short-circuiting due to the space advantage in the flow buffering technique.

Performance Gains: To understand the effectiveness of our schemes, we plot the average number of cycles taken to process a frame in each flow in Figure 5.16. This is the time between the invocation of first IP and completion of last IP in each flow. Note that, reducing the cycles per frame can lead to fewer frame drops. When we use our techniques with sub-framing, due to pipelining of intra-frame data across multiple IPs instead of sequentially processing one frame after another, we are able to substantially reduce the cycles per frame by 45% on average. We also observed that in A6-Skype application (which has multiple flows), through the use of sub-framing, the memory subsystem gets overwhelmed because, we allow more IPs to work at the same time. This is not the case in the base system. If IPs do not benefit from flow-buffers or IP request short-circuiting, the memory pressure is more than the baseline leading to some performance loss (17%).

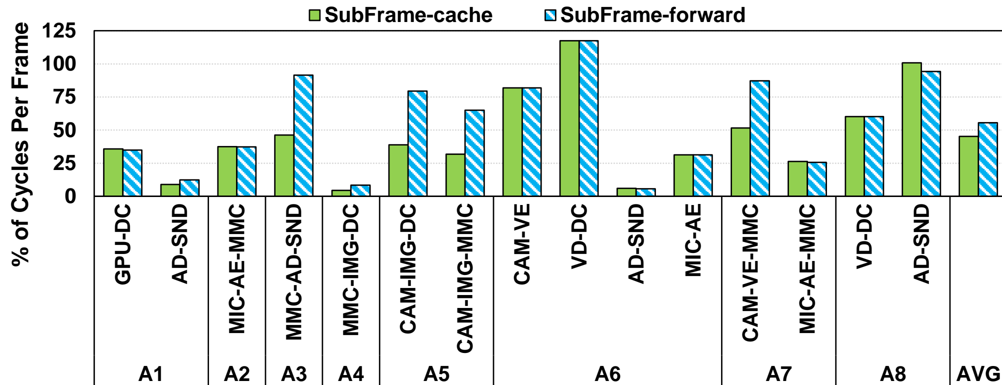


Fig. 5.16: Reduction in Cycles Per Frame in a flow normalized to Baseline (Lower the better).

Energy Gains: Energy efficiency is a very important metric in handhelds since they operate out of a battery (most of the time). Exact IP design and power states incorporated are not available publicly. As a proxy, we use the number of cycles an IP was active to correspond to the energy consumed when running the specific applications. In Figure 5.17, we plot the total number of active cycles consumed by an accelerator compared to the base case. We plot this graph only for accelerators as they are compute-intensive and hence, consume most of the power in a flow. On average, we observe 46% and 35% reduction in active cycles (going up to 80% in GPU) with our techniques, which translates to substantial system-level energy gains. With sub-framing, we also reduce the memory energy consumption by 33% due to (1) reduced DRAM accesses, and (2) memory spending more time in low-power mode. From the above results it can be concluded that sub-framing yields significant performance and power gains.

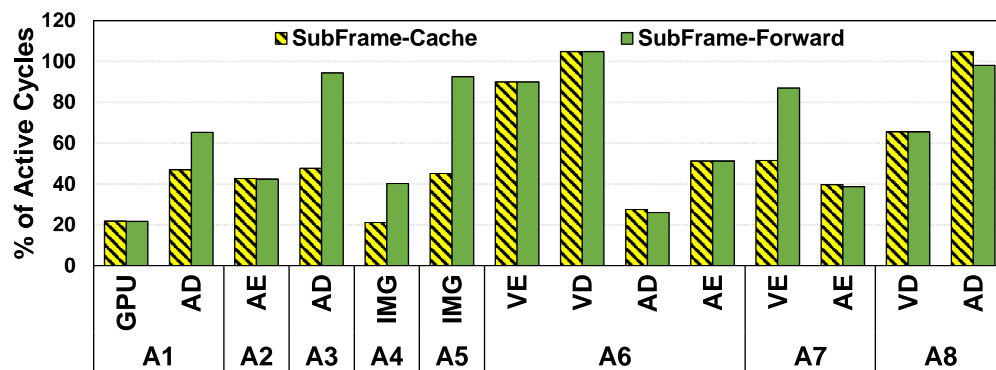


Fig. 5.17: Reduction in Number of Active Cycles of Accelerators (Lower the better).

Chapter 6

Cooperative Parallelization

6.1 Parallelization Approach

A high level graphical illustration of our approach is given in Figure 6.1. The solid arrows represent the control flow of the program and the dashed arrows represent the communication between threads. The left hand side of the figure illustrates the sequential execution of a program with a sequential loop to be parallelized. On the right hand side, parallel execution of the same program using our approach is given. With the help of runtime information from the helper thread, the main thread is able to parallelize the loop. In parallel execution, the main thread continues its work while the helper thread identifies the subproblems in the program.

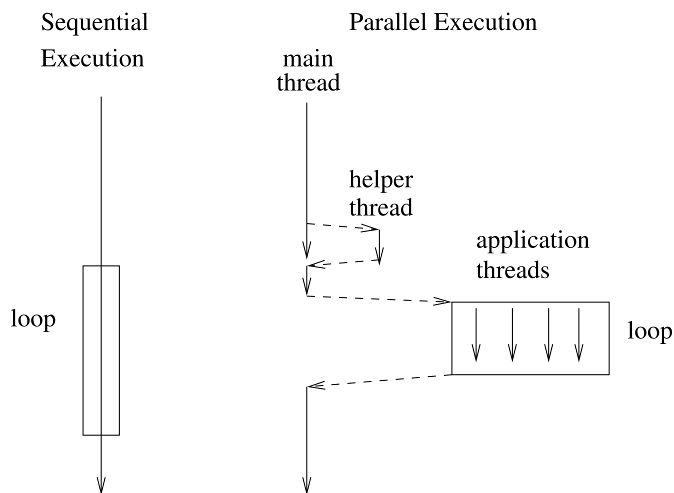


Fig. 6.1: A high level view of the proposed approach.

Figure 6.2 shows the high level code for the proposed execution model. The main thread signals the helper thread when the data structure becomes ready. The helper thread, waiting for the signal, senses the signal, and starts its job of finding independent subproblems in the data structure. After the helper thread's job is complete, it signals the main thread to notify the completion of finding of the subproblems. At this point, the main thread has access to multiple subproblems, and distributes these tasks to different application (worker) threads. The application threads that are waiting for a signal from the main thread start working on their part of the data structure. Once their jobs are

complete, they signal the main thread individually. The main thread, waiting for these signals, proceeds to merge the results from the individual threads.

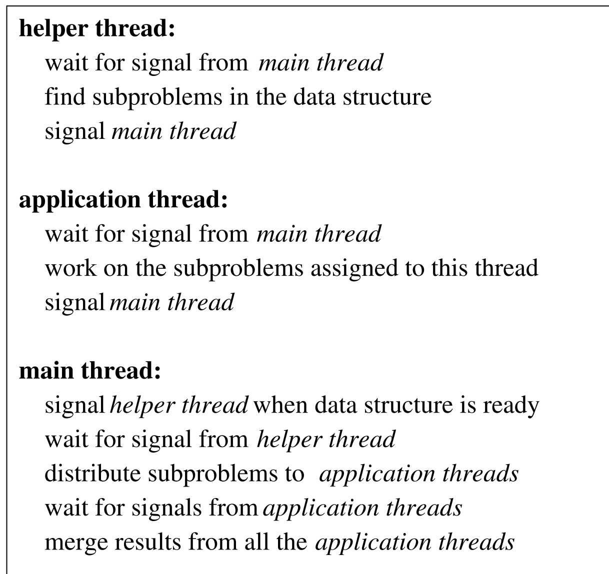


Fig. 6.2: High level code of a parallelized program using the proposed strategy.

6.1.1 Trees and Recursion

In this section, we describe how our approach handles tree based applications with recursion. Typically, in a tree based application, a tree data structure is constructed using the input and a function goes over the nodes of the tree and performs a computation at each node. This function frequently is a recursive function which is initially called on the root of the tree. The function performs some computation on the root node and in turn calls itself on each of its child nodes. Once the processing of the children is done, the function merges the results obtained from the function calls on the child nodes. Note that, the function calls on child nodes are calls to the same function, i.e., they are recursive function calls. Figure 6.3 graphically illustrates the process where the function called on the root node recursively calls itself to process the the left subtree and then the right subtree.

The key observation in the described scenario is that, the function calls on different subtrees can be independent. If this is the case, then the function need not wait for the call on the first child to complete in order to proceed with the next child. In this scenario, the only dependence is in merging the results obtained from the recursive calls which will be performed after all calls are completed. Therefore, the potential parallelism can be exploited by executing the function calls on child nodes in parallel. However, in order to launch these functions as threads executing in parallel, the contexts in which these functions will be called are needed. A *context* is defined as the set of parameters

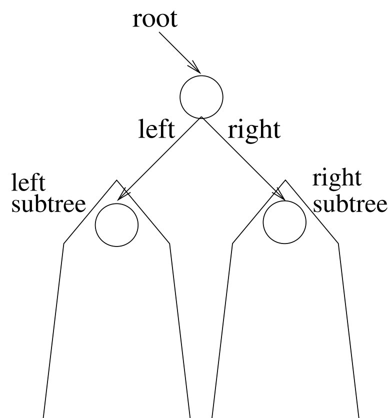


Fig. 6.3: Subproblems in a tree-based application.

that fully describes the environment that the thread will be launched in. In the tree example, if two threads will be launched for the two subtrees shown in Figure 6.3, then the pointers to the roots of the two subtrees are required to start the threads. Note that, in this binary tree, at a specific instance of the function, the target node has direct access only to its immediate children. As a result, the function can start a number of parallel tasks up to the number of children of its node. In order to be able to start more threads, deeper information over the tree is necessary.

We propose to move the task of finding the contexts for recursive function calls to the helper thread. The helper thread goes over the tree and finds the pointers to internal nodes along with the other parameters needed for the invocation of a function call. The helper thread tries to find a number of independent tasks to execute in parallel and sends this information to the main thread. It can find more number of tasks than the number of children to a node. The main thread distributes these independent tasks to application threads in a manner which balances the load over the available threads. Then, the main thread waits for the application threads to complete and merges their results.

As an example, consider the application *perimeter*, from the Olden benchmark suite Rogers et al. (1995), which works on a quad tree where each node has four children[1]. The *perimeter* function takes two parameters: a pointer to the root node of a tree and the size of the tree, as shown in Figure 6.4. The function first checks whether the target node is a leaf node. If this is the case, it performs a computation on that node; otherwise, it calls itself recursively on the child nodes (represented as *nw, ne, sw, se* fields) of the target node and half the size. The function adds the return values of all of recursive calls to compute the return value of the function.

Note that, in this application, the calls to the *perimeter* function on sibling nodes are independent and can be run in parallel. To parallelize such an application the main thread needs to know the contexts in which the functions on child nodes are called. In

[1]The details of the applications together with our evaluation methodology can be found in Section 6.3.

```
int perimeter (QuadTree tree, int size)
{
  int retval = 0;

  if (tree->color==grey) /*node has children */
  {
    retval += perimeter (tree->nw, size/2);
    retval += perimeter (tree->ne, size/2);
    retval += perimeter (tree->sw, size/2);
    retval += perimeter (tree->se, size/2);
  }
  else if (tree->color==black) /*node is a leaf node */
  {
    ... /* do something on the node*/
  }
  return retval;
}
```

Fig. 6.4: A function from perimeter Rogers et al. (1995), one of our tree-based applications.

this case, a function context comprises a node pointer and an integer. The main thread invokes a helper thread to identify the contexts. The helper thread starts at the root of the tree and finds the pointers to the four children along with the correct size values for each child node. Then, it returns these contexts to the main thread which initiates the application threads using these contexts, which have identical code as the original sequential program. Note that, if the application is to be parallelized with eight threads, then the helper thread goes one level deeper in the tree and identifies 16 second level pointers and size values. Then, the main thread must distribute the 16 tasks identified by the helper thread across the 8 application threads in a balanced fashion. Once all application threads complete execution, the main thread aggregates the results from individual jobs.

6.1.2 Linked Lists and Loops

In this section, we describe how our approach can be used to parallelize programs that use linked lists. In a typical linked list based application, once a linked list is constructed from the input, functions that operate on the linked list traverse the list sequentially using a *while* loop and perform computations on each node. If the computations performed on each node are independent, then the iterations of the loop are independent. In this case, the problem can be divided into subproblems, which are defined by the *sublists* of the original list. If the function has access the sublists of the list before starting the loop, then it can start working on different sublists in parallel. After the computations on the sublists are completed, the function can merge the results from the individual sublists to generate the final result.

Identification of the sublists is done by the helper thread which is invoked by the function before starting the loop. Once the helper thread completes its task, the function gains access to multiple sublists, and as a result, instead of executing the loop sequentially, it distributes the work on different sublists to different application threads. The application threads work on the sublists assigned to them and compute the local result from those sublists. Once all the application threads complete, the main thread then merges the results obtained from the individual sublists.

To further concretize the idea, consider the *em3d* application which operates on a singly linked list. The function *compute_nodes*, shown in Figure 6.5, contains an while loop which goes over the entire list and updates each node. Note that, the updates at a node are independent of the updates at all other nodes. To parallelize such a function, one needs to know the sublists in the linked list before starting the loop. Each sublist is defined by pointers to the start and end nodes in different parts of the linked list. The task of finding these pointers is carried out by the helper thread, which traverses the list and saves the pointers at particular intervals. Note that, the helper thread does not perform any computation but just traverses the list, which is why it can progress much faster than the main thread. Once the helper thread completes, the main thread gains access to different sublists of the linked list and distributes these sublists to different application threads. The application threads perform similar function as the original loop with the exception that they work on the sublists defined by the start and end nodes given by the main thread.

```
void compute_nodes (node_t * nodelist)
{
   int i;

   while ( nodelist != NULL )
   {
      for (i=0; i < nodelist->from_count; i++)
      {
         node_t *other_node = nodelist->from_nodes[i];
         double coeff = nodelist->coeffs[i];
         double value = other_node->value;

         nodelist->value -= coeff * value;
      }
      nodelist = nodelist->next
   }
}
```

Fig. 6.5: A function from em3d Rogers et al. (1995), one of our linked list-based applications.

6.2 Programmer Directives and Automation

Our automation framework consists of two components: (i) an effective way of expressing parallelism and (ii) a method to generate parallel code from the sequential code automatically. We implement the first component by designing simple compiler directives that are used to annotate the target (sequential) program code. The second component is realized by a source-to-source compiler that converts user annotated program code to parallel code. In this section, we give detailed description of these two components.

6.2.1 Programmer Directives

The first problem in parallelizing applications is the identification of independent tasks in a program. In regular programs that mostly use arrays and for loops to access elements of arrays, this can be done by data dependence analysis Maydan et al. (1991, 1993); Goff et al. (1991); Pugh (1991); Wolfe (1995). However, in an irregular program involving pointer-accessed dynamic data structures, recursion, and while loops, it is a very difficult task to identify data dependences automatically. Instead of performing data dependence analysis, we take an alternative approach and obtain the parallelism information from the programmer. In order to capture this information from the programmer, we need well defined compiler directives that express parallelism in the program. The design goals of these directives are that, they should be compact with minimal essential information and simple enough to be easily used. Another use of the directives is to dictate the features of the parallel program such as the number of threads. We propose two types of directives, one for tree based applications and one for linked list based applications, which are shown in Figure 6.6.

```
#parallel tree function (threads) (degree) (structure) {subproblems} val

#parallel llist function (threads) (structure) subproblem number
```

Fig. 6.6: Programming directives to express parallelism. Note that, *val* and *number* are optional fields.

A tree based application typically consists of a recursive function in which, calls to the same function in different contexts (defined by the parameters) can be parallelized. The directive conveys exactly this information, where the first two fields indicate that the directive is a parallelization directive and the program operates over a tree structure. This information is used to select the type of helper thread to create for the application. The *function* field indicates the name of the function to be parallelized and the *threads* field indicates the number of worker threads that will be created in the parallel version of the code. Note that, the number of threads information is critical in the sense that the overall speedup obtained from our framework heavily depends on this number. This field gives the user the capability to decide on the appropriate number of threads in the

parallel program considering the number of processing elements available. The *degree* field indicates the number of children that each node in the tree has, which is used to customize the helper thread according to the specific tree structure used in the program. The *structure* and *subproblems* fields indicate the name of the structure and the name of the child nodes in the tree structure, which are used to identify the subtrees that will be searched to detect parallel tasks. Finally, the *val* field indicates the field of the tree node on which the function performs its computation. This field is optional and is required only when the function is accumulating results from the individual subtrees.

It should be noted that some of the fields, such as *degree* and *subproblems*, can be obtained from static analysis of the node structure of the tree. However, in practice, a node structure can have many fields and it is a difficult task for an automated system to discover the meaning of each field. As a result, instead of employing complicated analyses for this purpose, we ask the programmer to identify these fields. For example, consider the following directive that targets the *perimeter* application:

#parallel tree perimeter (2) (4) (QuadTree) {nw, ne, sw, se}

This directive means that the target program is a tree based application, where the *perimeter* function will be parallelized. Furthermore, the programmer indicated that two threads will be used to parallelize the application. The name of the tree structure is *QuadTree* and each node has four children with the names *nw, ne, sw* and *se*.

Figure 6.6 also shows our second type of directive. This directive is used to express parallelism in linked list-based applications, which typically consist of a loop traversing the elements of the list. The *llist* field indicates that the program to be parallelized is a linked list-based application. Similar to the previously described tree-based directive, the parameter *structure* defines the name of the structure corresponding to a node of the linked list. The *subproblem*, on the other hand, identifies the field of a linked list node that points to the next node in the list. The estimated number of nodes in the linked list is given by the *number* field, which is an optional field that in turn determines the size of the sublists the linked list will be divided into. Note that, if the provided number is too small, then there will be a small number of nodes per sublist and high number of sublists. This in turn increases the relative cost of distributing the computation to the worker threads and later merging their results. Therefore, providing a correct estimate generates efficient parallel code with less overhead. Along with this directive, the programmer also provides another directive (#parallel) just before the loop which is to be parallelized. This additional directive is used to identify the target loops in the function. Consider the following directive that targets the *em3d* application:

#parallel llist compute_nodes (2) (node_t *) next 10000

This directive indicates that the program is a linked list based application and the function *compute_nodes* contains a loop that can be executed in parallel using two threads. The name of the list node structure is *node_t* and the next item in the list can be accessed using the *next* field of the node. The programmer also provided an estimate of ten thousand elements in the list.

6.2.2 Automation

Once the code is annotated with the programmer directives (hints) described in the previous section, these directives need to be parsed and leveraged to generate the parallel program. In order to achieve this, we implemented a source-to-source translator that takes the original program with parallelism directives as input, attaches a helper thread to it, and generates a parallel version of the program that receives vital parallelization information from the helper thread. Source-to-source translation is actually performed while parsing the high level source code of the original program. For this purpose, we modified the C language grammar to include the programmer directives and provided this grammar to a parser generator. The translator generated from the parser generator parses the source code of the program and creates the parallel program simultaneously. Figure 6.7 shows the high level view of this process. When the parser detects a parallelization directive, it creates the functions for helper thread, application threads, and the modified parallel function. The automation of program parallelization is made possible due to similarity in steps taken in creating the parallel programs.

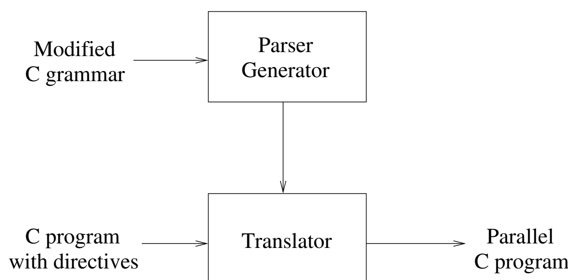


Fig. 6.7: High level view of automation.

In a tree based application, the helper thread needs to find the contexts of multiple function calls, which include pointers to the subtrees. The helper thread traverses the tree to find pointers to a number of subtrees based on the number of parallel threads. The number of discovered subtrees should be equal to or greater than the number of threads so that each thread gets at least one subtree to work on. In order to discover more contexts, the helper thread scans more levels of the tree. Note that, the code for the helper thread is identical for all tree based applications except for the slight differences caused by the differences in the degrees of the trees. The application thread code is similar to the original function, which recursively calls itself over all child nodes. In some cases, the application thread has to work on multiple contexts assigned to it by the main thread. Note that, the application threads should call the original function to perform the computations on the child nodes. The original function is modified to invoke the helper thread, receive the context information from the helper thread, distribute these contexts to different application threads, and merge their results. A parallel function in a tree based application should perform these operations. The code in the parallelized function is similar for all the tree based applications except for the merging part.

In a linked list based application, the helper thread is invoked to find sublists of the linked list, identified by their pointers to start and end nodes. The helper thread finds pointers to a number of sublists based on the number of threads to be used to parallelize the application. The number of sublists should at least be equal the number of threads. The code for the helper thread is created by finding a *slice* of the loop over the loop control variable. This results in a loop which contains only the statements which traverse the loop in the same way as the original loop. Along with these statements, we add the code that saves the pointers at particular intervals which define the boundaries of the sublists. The application thread has to do exactly what the original loop was doing but on a sublist of the list. The application threads contain similar code as the original loop with the exception of loop boundaries which are dependent on the sublist on which the thread is working on. The original function is also modified; instead of executing the loop, it invokes the helper thread, distributes the sublists to different application threads, and merges the results from individual application threads. A parallel function in a list based application should perform these operations. The code in the parallelized function is similar for all the list based applications except for the merging part.

6.3 Experimental Evaluation

In this section, we present the experimental setup and evaluation of our automated parallelization approach.

6.3.1 Platform

We target shared memory multicore systems for this work. The details of the platform used for our experiments are provided in Table 6.1.

Hardware	Dual Socket Intel Xeon E7450 @ 2.40GHz 2 Hexa-cores, 12 cores in total 12MB L2 Cache per chip 4GB RAM
Operating System	64-bit Linux with 2.6.29.6 kernel
Compiler	gcc 4.1.2

Table 6.1: Hardware and Software configuration of the experimental evaluation platform.

6.3.2 Benchmarks

We used the Olden benchmark suite Rogers et al. (1995) to evaluate our approach. The suite consists of nine applications which are based on trees and linked lists, and are written in C programming language. Out of the nine applications, three of them

have data dependences between their statements, which prevent parallel execution. We parallelized the remaining six applications and *otter* Laboratory (2009) which is a linked list-based application. The details of these applications are shown in Table 6.2. For each application, the second column gives the brief description and the third column indicates the type of the data structure used in the application. We profiled the applications using *gprof*, and the input set given in column four. The functions we parallelized and their contribution to the overall execution times of the applications are also given in columns five and six, of the same table respectively. Note that, the higher the execution time fraction, the better overall speedup over the whole application when that function is parallelized.

Benchmark	Description	Data Structure	Input Size	Core Function	Execution Time
bisort	Bitonic Sorting	Binary Tree	100,000 numbers	bisort	50%
treeAdd	Adding Numbers in Nodes	Binary Tree	22 levels	treeAdd	57%
tsp	Traveling Salesman	Binary Tree	10,000 cities	tsp	90%
perimeter	Perimeters of Regions in Images	Quad Tree	11 levels	perimeter	65%
mst	Minimum Spanning Tree	Singly Linked List	4096 nodes	BlueRule	91%
em3d	Electron Microscope Tomography	Singly Linked List	6,000 n_nodes & 100 d_nodes	compute_nodes	83%
otter	Theorem Proving Software	Singly Linked List	twoval.in	find_lightest_cl	20%

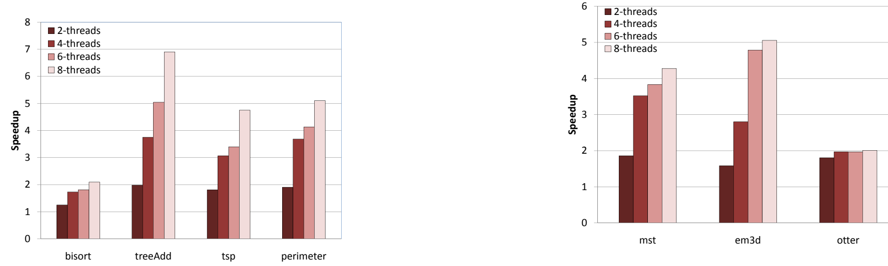
Table 6.2: Benchmark applications and their important properties.

We identified the parallelism available in each application by manual inspection and added the appropriate directives (described earlier in Section 6.2.1) to the original programs. These modified programs are processed using the translator described in the previous section, which understands the directives and automatically generates the parallel C programs. During our experiments, we bound each thread of the parallelized application to a different core in the underlying platform. We measured the running times of each of the parallelized applications and compared them with the sequential execution times.

6.3.3 Results

The results we obtained with the proposed approach are shown in Figures 6.8a and 6.8b. The first graph shows the speedups obtained on tree based applications and the second graph shows the speedups obtained on linked list based applications. In both the figures, the x-axis represents different applications considered, and the y-axis gives the speedups obtained on the particular function or a loop that is parallelized. Each

bar in the graph corresponds to a different number of application threads used in the parallel program. In most of the applications, we obtained almost linear speedups. The speedup in *bisort* is not significant because of the costly merge operations occurring after the parallel computations. The speedup in *otter* is not significant due to the overheads in our approach. These overheads are discussed in detail in the next section.



(a) Speedups with the tree-based applications

(b) Speedups with the linked list-based applications

Fig. 6.8: Speedups obtained with our approach

Although our technique can handle the generation of parallel code, an exception in our implementation is that, in programs with complex merge operations, it cannot produce accurate code to merge the results from different application threads. If the program is just finding the sum or product of the results from individual threads, that is automated in our translator. The difficulty in handling complex merge operations is that, in a recursive environment, the merges in deeper levels of recursion should happen before the merges in higher levels. Since we are unrolling the recursion in a way, we need a mechanism to determine the order in which the results are to be merged. This feature is not yet fully automated in our translator. Instead, in our current implementation, the user of the tool has to examine the generated code specifically at the merging part and do any required modifications manually. We believe that, this does not arise frequently in general programs. Even in cases it arises, the programmer should be able to easily add the code needed for the merging. In the seven benchmarks we tested our approach, three of them (bisort, tsp, and otter) required slight modification in the merging part.

Chapter 7

Future Work

7.1 Cooperative Prefetching

Current cache prefetcher designs are not well suited for S-NUCA based CMPs. In such a system, each L2 bank gets requests from multiple L1s. A traditional L2 prefetcher monitors the L2 misses happening from that bank and maintains miss streams at the L2 bank. When each L2 bank is serving multiple L1s at the same time it is very difficult to detect streams at L2 banks. We are working on a new stream prefetcher design for S-NUCA based systems. We maintain the streams corresponding to an application at the corresponding L1 bank instead of multiple L2 banks.

Core-side prefetcher has the following characteristics (i) High accuracy, (ii) Agnostic to memory-state and, (iii) On-chip resource pollution. Whereas, Memory-side prefetcher has the following characteristics (i) Relatively low accuracy, (ii) Opportunistic, (iii) Memory-state aware and, (iv) No on-chip resource pollution. In our memory-side prefetching work we showed that both the prefetchers can work together to yield better benefits than either one of them individually. However, there are some smart ways to combine the two where both of them work together instead of independently. For example, memory-side prefetcher can bring the data to prefetch buffer at the memory controller and core-side prefetcher prefetches only from the prefetch buffer without disturbing the off-chip memory. We are currently working on multiple ways to integrate these two prefetchers to achieve maximum benefits.

7.2 Mobile Memory Systems

Many of the memory issues discussed in this dissertation are in the context of high performance computing. Similar issues can occur in other platforms like mobiles or system on chip (SOCs). Typically, SOCs use independent accelerators for specific tasks like graphics, video encoding/decoding etc. Accelerators coordinate with CPU and communicate through memory to accomplish tasks. Power is a major concern in mobile platform. As a result, low power memory designs (LP-DDR) are used in such systems. The problem with LP-DDR is it runs at half the frequency compared to a DDR memory and therefore has half the bandwidth. With modern mobile applications, the bandwidth requirements of accelerators can overwhelm the available bandwidth from memory. The parallelism vs. locality tradeoff in these scenarios will be very different than the results presented in this dissertation which are got on high performance systems. As a future work we want to explore this area more.

Chapter 8

Concluding Remarks

Memory wall challenge is an important challenge faced by current generation CMPs. In this dissertation we looked at this problem from the parallelism vs locality perspective. We started with a characterization study on parallelism and locality dimensions at different levels of the memory hierarchy. From the findings of that study, we proposed a dynamic memory migration technique to optimize both locality and parallelism in the memory subsystem. We presented the major challenges faced by traditional cache prefetchers in modern CMPs. We showed how memory prefetching can take advantage of the memory locality and prefetch opportunistically, leading to better efficiency than traditional cache prefetchers.

Current mobile computing platforms are performance bottlenecked by the memory subsystem, especially for media and graphics applications. These applications have a lot of temporal locality of data between producer and consumer cores/IPs. We proposed to break the application frames into sub-frames in order to exploit the memory locality and reduce the memory bandwidth requirements in such systems. All of these techniques try to reduce the difference in speed between the processor and memory and abate the memory wall problem.

Appendix

Publications

A.1 Significant Publications

[ICCAD 2011]
Praveen Yedlapalli, Emre Kultursay, Mahmut Kandemir, *Cooperative Parallelization*, In 30th International Conference on Computer Aided Design (ICCAD)

[PACT 2013]
Praveen Yedlapalli, Jagadish Kotra, Emre Kultursay, Chita Das, Mahmut Kandemir, Anand Sivasubramaniam, *Meeting Midway: Improving DRAM Performance and Off-Chip Latencies with Memory-Side Prefetching*, In 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)

[MICRO 2014]
Praveen Yedlapalli, Nachiappan Chidambaram, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut Kandemir, Chita Das, *Short-Circuiting Memory Traffic in Handheld Platforms*, In 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

[IISWC 2015 - Submitted]
Praveen Yedlapalli, Mahmut Kandemir, Chita Das, Anand Sivasubramaniam, *Quantifying and Exploiting Parallelism-Locality Tradeoff across Memory Hierarchy*, In IEEE International Symposium on Workload Characterization (IISWC)

A.2 Other Publications

[HPCA 2015]
Nachiappan Chidambaram, Praveen Yedlapalli, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut Kandemir, Ravishankar Iyer, Chita Das *Domain Knowledge Based Energy Management in Handhelds*, In 21st IEEE Symposium on High Performance Computer Architecture (HPCA)
[IISWC 2014 - Short Paper]
Yang Ding, **Praveen Yedlapalli**, Mahmut Kandemir, *QoS Aware Dynamic Time-Slice Tuning*, In IEEE International Symposium on Workload Characterization (IISWC)
[IISWC 2014 - Short Paper]
Umut Orhan, Umut Orhan, **Praveen Yedlapalli**, Mahmut Kandemir, Ozcan Ozturk *A Cache Topology-Aware Multi-Query Scheduler for Multicore Architectures*, In IEEE International Symposium on Workload Characterization (IISWC)

[PACT 2014]
Wei Ding, Mahmut Kandemir, Diana Guttman, Adwait Jog, Chita R. Das, **Praveen Yedlapalli** *Trading Cache Hit Rate for Memory Performance*, In 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)
[SIGMETRICS 2014]
Nachiappan Chidambaram, **Praveen Yedlapalli**, Niranjan Soundararajan, Mahmut Kandemir, Anand Sivasubramaniam, Chita Das *GemDroid: A Framework to Evaluate Mobile Platforms*, In International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)
[CGO 2013]
Wei Ding, Yuanrui Zhang, Mahmut Kandemir, Jithendra Srinivas, **Praveen Yedlapalli**, *Locality-Aware Mapping and Scheduling for Multicores*, In International Symposium on Code Generation and Optimization (CGO)
[DATE 2010]
Yuanrui Zhang, Lanping Deng, **Praveen Yedlapalli**, Sai Muralidhara, Nikos Pitsianis, Xiaobai Sun, Mahmut Kandemir, Chaitali Chakrabarti , *A special-purpose compiler for function evaluation code generation*, In Design, Automation and Test in Europe (DATE)

Bibliography

A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, Mar. 1995.

C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *SIGARCH Comput. Archit. News*, 2002.

M. Cade and A. Qasem, "Balancing locality and parallelism on shared-cache multicore systems," in *HPCC*, 2009.

J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler, "A nuca substrate for flexible cmp cache sharing," *Parallel and Distributed Systems, IEEE Transactions on*, 2007.

O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *HPCA*, 2003.

M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing dram locality and parallelism in shared memory cmp systems." in *HPCA*, 2012.

K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: Increasing dram efficiency with locality-aware data placement," in *ASPLOS*, 2010.

Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (salp) in dram," in *ISCA*, 2012.

A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking dram design and organization for energy-constrained multi-cores," in *ISCA*, 2010.

R. Das, S. Eachempati, A. Mishra, V. Narayanan, and C. R. Das, "Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps," in *HPCA*, 2009.

D. Park, S. Eachempati, R. Das, A. K. Mishra, Y. Xie, N. Vijaykrishnan, and C. R. Das, "Mira: A multi-layered on-chip interconnect router architecture," in *ISCA*, 2008.

C. Kim, D. Burger, and S. Keckler, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *Micro, IEEE*, 2003.

C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn, "Profile-guided post-link stride prefetching," in *ICS*, 2002.

D. Ortega, E. Ayguadé, J.-L. Baer, and M. Valero, "Cost-effective compiler directed memory prefetching and bypassing," in *PACT*, 2002.

Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *PLDI*, 2002.

D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *ISCA*, 1997.

G. Liu, Z. Huang, J.-K. Peir, X. Shi, and L. Peng, "Enhancements for accurate and timely streaming prefetcher," *The Journal of ILP*, vol. 13, Jan. 2011.

S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA*, 2007.

X. Zhuang and H.-H. S. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," in *ICPP*, 2003.

T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes," in *ISCA*, 1994.

E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multicore systems," in *MICRO*, 2009.

C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Pacman: prefetch-aware cache management for high performance caching," in *MICRO*, 2011.

C. J. Hughes and S. V. Adve, "Memory-side prefetching for linked data structures for processor-in-memory systems," *Journal of PDC*, 2005.

I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *MICRO*, 2006.

Y. Solihin, J. Lee, and J. Torrellas, "Correlation prefetching with a user-level memory thread," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 6, Jun. 2003.

C.-L. Yang and A. R. Lebeck, "Push vs. pull: data movement for linked data structures," in *ICS*, 2000.

"Vision Statement: How People Really Use Mobile," January-February 2013. [Online]. Available: http://hbr.org/2013/01/how-people-really-use-mobile/ar/1

R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov, "System-on-chip: Reuse and integration," *Proceedings of the IEEE*, 2006.

Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *HPCA*, 2013.

K.-B. Lee and T.-S. Chang, *Essential Issues in SoC Design Designing - Complex Systems-on-Chip.* Springer, 2006, ch. SoC Memory System Design.

N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: A measurement study and implications for network applications," in *IMC*, 2009.

Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, “Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,” in *MICRO*, 2010.

Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, “ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers,” in *HPCA*, 2010.

R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, “Aergia: Exploiting packet latency slack in on-chip networks,” in *ISCA*, 2010.

C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, “The amd opteron processor for multiprocessor servers,” *IEEE Micro*, 2003.

P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, “Cache hierarchy and memory subsystem of the amd opteron processor,” *IEEE micro*, 2010.

A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, “Power management architecture of the 2nd generation intel® core microarchitecture, formerly codenamed sandy bridge,” 2011.

M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, “A fully integrated multi-cpu, gpu and memory controller 32nm processor,” in *ISSCC*, 2011.

IBM, “Cell broadband engine,” 2009, http://www.research.ibm.com/cell/.

Intel, “Xeon processor,” 2009, http://www.intel.com/p/en_US/products/server/processor.

AMD, “Opteron processor,” 2009, http://www.amd.com/us/products/server/processors/.

L. Lamport, “The parallel execution of do loops,” *Commun. ACM*, vol. 17, no. 2, pp. 83–93, 1974.

M. J. Wolfe, *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., 1995.

U. K. Banerjee, *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

——, *Loop Parallelization.* Kluwer Academic Publishers, 1994.

D. S. Steve Scheirey, “Sensor fusion, sensor hubs and the future of smartphone intelligence,” ARM, Tech. Rep., 2013.

J. Engwell, “The high resolution future retina displays and design,” Blurgroup, Tech. Rep., 2013.

J. Y. C. Engwell, “Gpu technology trends and future requirements,” Nvidia Corp., Tech. Rep.

G. E. Blelloch, J. Fineman, P. B. Gibbons, and H. V. Simhadri, “Program-centric cost models for locality,” in *MSPC*, 2013.

S. K. Singhai, Kathryn, and S. Mckinley, "A parametrized loop fusion algorithm for improving parallelism and cache locality," *Computer Journal*, 1997.

M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *PACT*, 2010.

M. Chaudhuri, "Pagenuca: Selected policies for page-grain locality management in large shared chipmultiprocessor caches," in *HPCA*, 2009.

N. Eisley, L.-S. Peh, and L. Shang, "Leveraging on-chip networks for data cache migration in chip multiprocessors," in *PACT*, 2008.

G. Chen, G. Chen, O. Ozturk, and M. Kandemir, "Exploiting inter-processor data sharing for improving behavior of multi-processor socs," in *ISVLSI*, 2005.

L. Xue, M. Kandemir, G. Chen, and T. Yemliha, "Spm conscious loop scheduling for embedded chip multiprocessors," in *ICPADS*, 2006.

M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *ASPLOS*, 2002.

M. Kandemir, J. Ramanujam, and A. Choudhary, "Exploiting shared scratch pad memory space in embedded multiprocessor systems," in *DAC*, 2002.

V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for mpsoc architectures," in *CASES*, 2006.

P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang, "Mapping of applications to mpsocs," in *CODES+ISSS*, 2011.

A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, "Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms," *J. Syst. Archit.*, 2010.

A. Coskun, T. Rosing, and K. Whisnant, "Temperature aware task scheduling in mpsocs," in *DATE*, 2007.

B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable sdram memory controller," in *CODES+ISSS*, 2007.

M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *DAC*, 2012.

H. b. T. Khan and M. K. Anwar, "Quality-aware Frame Skipping for MPEG-2 Video Based on Inter-frame Similarity," Malardalen University, Tech. Rep.

S. Fenney, "Texture compression using low-frequency signal modulation," in *HWWS*, 2003.

H. Shim, N. Chang, and M. Pedram, "A compressed frame buffer to reduce display power consumption in mobile systems," in *ASP-DAC*, 2004.

K. Han, A. Min, N. Jeganathan, and P. Diefenbaugh, "A hybrid display frame buffer architecture for energy efficient display subsystems," in *ISLPED*, 2013.

A. Gutierrez, R. Dreslinski, A. Saidi, C. Emmons, N. Paver, T. Wenisch, and T. Mudge, "Full-system analysis and characterization of interactive smartphone applications," in *IISWC*, 2011.

Y. Wang, B. Krishnamachari, Q. Zhao, and M. Annavaram, "Markov-optimal sensing policy for user state estimation in mobile devices," in *IPSN*, 2010.

B. Diniz, D. O. G. Neto, W. M. Jr., and R. Bianchini, "Limiting the power consumption of main memory." in *ISCA*, 2007.

J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *HPCA*, 1999.

W.-f. Lin, "Reducing DRAM latencies with an integrated memory hierarchy design," in *HPCA*, 2001.

S. Ryoo, S.-Z. Ueng, C. I. Rodrigues, R. E. Kidd, M. I. Frank, and W.-M. W. Hwu, *Automatic Discovery of Coarse-Grained Parallelism in Media Applications*. Springer-Verlag, 2007, pp. 194–213.

G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *SIGPLAN Not.*, vol. 44, no. 6, pp. 177–187, 2009.

G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2005, pp. 105–118.

E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proceedings of the sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2008, pp. 114–123.

N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007, pp. 49–59.

F. Aleen and N. Clark, "Commutativity analysis for software parallelization: letting program transformations see the big picture," *SIGPLAN Not.*, vol. 44, no. 3, pp. 241–252, 2009.

M. Rinard and P. Diniz, "Commutativity analysis: A new analysis framework for parallelizing compilers," University of California at Santa Barbara, Tech. Rep., 1996.

S. Rus, M. Pennings, and L. Rauchwerger, "Sensitivity analysis for automatic parallelization on multi-cores," in *Proceedings of the 21st Annual International Conference on Supercomputing.* ACM, 2007, pp. 263–273.

S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid analysis: static & dynamic memory reference analysis," *Int. J. Parallel Program.*, vol. 31, no. 4, pp. 251–283, 2003.

L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *SIGPLAN Not.*, vol. 30, no. 6, pp. 218–232, 1995.

F. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD test: Speculative parallelization of partially parallel loops," in *Proceedings of the 16th International Symposium on Parallel and Distributed Processing.* IEEE Computer Society, 2002, p. 20.

M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2007, pp. 69–84.

P. I. Standard, "Openmp: A proposed industry standard api for shared memory programming," 2009, http://openmp.org/wp/.

B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "DyC: an expressive annotation-directed dynamic compiler for C," *Theor. Comput. Sci.*, vol. 248, no. 1-2, pp. 147–199, 2000.

L. Rauchwerger and D. A. Padua, "Parallelizing while loops for multiprocessor systems," in *Proceedings of the 9th International Symposium on Parallel Processing.* IEEE Computer Society, 1995, pp. 347–356.

M. Gupta, S. Mukhopadhyay, and N. Sinha, "Automatic parallelization of recursive procedures," *Int. J. Parallel Program.*, vol. 28, no. 6, pp. 537–562, 2000.

R. Rugina and M. C. Rinard, "Pointer analysis for structured parallel programs," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 1, pp. 70–116, 2003.

B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *Proceedings of the International Symposium on Code Generation and Optimization.* IEEE Computer Society, 2005, pp. 291–302.

J. Da Silva and J. G. Steffan, "A probabilistic pointer analysis for speculative optimizations," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 2006, pp. 416–425.

J. Steffan and T. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture.* IEEE Computer Society, 1998, pp. 2–13.

D. Bruening, S. Devabhaktuni, and S. Amarasinghe, "Softspec: Software-based speculative parallelism," in *In 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 1998.

Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A cost-driven compilation framework for speculative parallelization of sequential programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2004, pp. 71–81.

H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

E. Raman, N. Vachharajani, R. Rangan, and D. I. August, "Spice: speculative parallel iteration chunk execution," in *Proceedings of the sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization.* ACM, 2008, pp. 175–184.

C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: an infrastructure for speculative threading based on precomputation slices," *SIGPLAN Not.*, vol. 40, no. 6, pp. 269–279, 2005.

J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 1–12, 2000.

V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *MICRO*, 2013.

A. Mishra, O. Mutlu, and C. Das, "A heterogeneous multiple network-on-chip design: An application-aware approach," in *DAC*, 2013.

R. Das, S. Narayanasamy, S. K. Satpathy, and R. G. Dreslinski, "Catnap: Energy proportional multiple network-on-chip," in *ISCA*, 2013.

Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO*, 2000.

Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ISCA*, 2008.

S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *ICS*, 2004.

M. Karlsson, F. Dahlgren, and P. Stenstrom, "A prefetching technique for irregular accesses to linked data structures," in *HPCA*, 2000.

D. K. Poulsen and P.-C. Yew, "Data prefetching and data forwarding in shared memory multiprocessors," in *ICPP*, 1994.

A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, Dec. 1978.

E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multicore systems," in *ISCA*, 2011.

P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "SIMICS: A full system simulation platform," *Computer*, 2002.

M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacets general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, 2005.

K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors," in *ISPASS*, 2001.

R. Hegde, "Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers," *Intel*, 2008.

C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware DRAM controllers," in *MICRO*, 2008.

Micron, "DDR3 Power Calculator." 2009.

A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. Kandemir, and C. R. Das, "GemDroid: A Framework to Evaluate Mobile Platforms," in *SIGMETRICS*, 2014.

Google. (2013) Android sdk - emulator. [Online]. Available: http://developer.android.com/

P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *CAL*, 2011.

R. SoC, "Rk3188 multimedia codec benchmark," 2011.

Samsung, "Samsung galaxy s5," 2014. [Online]. Available: http://www.samsung.com/global/microsite/galaxys5/

Apple, "Apple iphone 5s." [Online]. Available: https://www.apple.com/iphone/

P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," Technical Report 2001/2, Compaq Computer Corporation, Tech. Rep., 2001.

Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," in *ACM SIGPLAN Notices*, 1999.

A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *POPL*, 1997.

T. Sha, M. M. K. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *MICRO*, 2005.

G. H. Loh, R. Sami, and D. H. Friendly, "Memory bypassing: Not worth the effort," in *WDDD*, 2002.

S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA*, 2000.

H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the h. 264/avc standard," *IEEE Transactions on Circuits and Systems for Video Technology*, 2007.

Google, "Android HAL." [Online]. Available: https://source.android.com/devices/index.html

S. T. Report, "Wqxga solution with exynos dual," 2012.

D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 1991, pp. 1–14.

D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array-data flow analysis and its use in array privatization," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 1993, pp. 2–15.

G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," *SIGPLAN Not.*, vol. 26, no. 6, pp. 15–29, 1991.

W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing.* ACM, 1991, pp. 4–13.

A. N. Laboratory, "Otter," 2009, http://www.mcs.anl.gov/research/projects/AR/otter/.

Vita

I am a Ph.D. Candidate in the Computer Science and Engineering Department at Penn State University. I am working in the area of Computer Architecture with Dr. Mahmut Kandemir for my PhD. I also collaborate with Dr. Anand Sivasubramaniam and Dr Chita R. Das. Currently, I am working on memory and network optimizations for NOC based CMPs. I mainly focus on optimizing the memory controller architecture in order to improve the memory bandwidth available for the processor. Prior to coming here, I received my Master of Technology (M.Tech.) in Computer Science and Engineering in 2008 from Indian Institute of Technology Kanpur. I received my Bachelor of Technology (B.Tech.) in Computer Science and Engineering in 2006 from Andhra University.

I worked as an intern for VMware at Palo Alto, CA in the summer of 2013. I worked in the ESX performance team developing new in-memory compression techniques for the ESX hypervisor. I worked as an intern for Intel Corporation at Champaign, IL for a major part of 2011. I worked in the Thread Checker team developing a new light weight version of the tool to enable its usage on embedded platforms.