REDUCING PROCESSOR-MEMORY PERFORMANCE GAP AND IMPROVING NETWORK-ON-CHIP THROUGHPUT

A DISSERTATION SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

By

Naveed Ul Mustafa

February 2019

REDUCING PROCESSOR-MEMORY PERFORMANCE GAP AND IMPROVING NETWORK-ON-CHIP THROUGHPUT
By Naveed Ul Mustafa
February 2019

We certify that we have read this dissertation and that in our opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Özcan Öztürk(Advisor)

Süleyman Tosun

Uğur Güdükbay

Muhammet Mustafa Özdal

Kayhan Mustafa İmre

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

REDUCING PROCESSOR-MEMORY PERFORMANCE GAP AND IMPROVING NETWORK-ON-CHIP THROUGHPUT

Naveed Ul Mustafa
Ph.D. in Computer Engineering
Advisor: Özcan Öztürk
February 2019

Performance of computing systems has tremendously improved over last few decades primarily due to decreasing transistor size and increasing clock rate. Billions of transistors placed on a single chip and switching at high clock rate result in overheating of the chip. The demand for performance improvement without increasing the heat dissipation lead to the inception of multi/many core design where multiple cores and/or memories communicate through a network on chip. Unfortunately, performance of memory devices has not improved at the same rate as that of processors and hence become a performance bottleneck. On the other hand, varying traffic pattern in real applications limits the network throughput delivered by a routing algorithm.

In this thesis, we address the issue of reducing processor-memory performance gap in two ways: First, by integrating improved and newly developed memory technologies in memory hierarchy of a computing system. Second, by equipping the execution platform with necessary architectural features and enabling its compiler to parallelize memory access instructions. We also address issue of improving network throughput by proposing a selection scheme that switches routing algorithm of an NoC with changing traffic pattern of an application.

We present integration of emerging non-volatile memory (NVM) devices in memory hierarchy of a computing system in the context of database management systems (DBMS). To this end, we propose modifications in storage engine (SE) of a DBMS aiming at fast access to data through bypassing the slow disk interfaces while maintaining all the functionalities of a robust DBMS. As a case study, we modify the SE of PostgreSQL and detail the necessary changes and challenges such modifications entail. We evaluate our proposal using a comprehensive emulation platform. Results indicate that our modified SE reduces query execution time by up to 45% and 13% when compared to disk and NVM storage, with average reductions of 19% and 4%, respectively. Detailed analysis of these results shows that our modified SE suffers from data readiness problem. To solve this, we develop a general purpose library that employs helper threads to prefetch data from NVM hardware via a simple application program interface (API). Our library further improves query execution time for our modified SE when compared to disk and NVM storage by up to 54% and 17%, with average reductions of 23% and 8%, respectively.

As a second way to reduce processor-memory performance gap, we propose a compiler optimization aiming at reduction of memory bound stalls. The proposed optimization generates efficient instruction schedule through classification of memory references and consists of two steps: affinity analysis and affinity-aware instruction scheduling. We suggest two different approaches for affinity analysis, i.e., source code annotation and automated analysis. Our experimental results show that application of annotation-based approach on a memory intensive program reduces stall cycles by 67.44%, leading to 25.61% improvement in execution time. We also evaluate automated-analysis approach using eleven different image processing benchmarks. Experimental results show that automated-analysis reduces stall cycles, on average, by 69.83%. As all benchmarks are both compute and memory-intensive, we achieve improvement in execution time by up to 30%, with a modest average of 5.79%.

In order to improve network throughput, we propose a selection scheme that switches routing algorithm with changing traffic pattern. We use two selection strategies: static and dynamic selection. While static selection is made off-line, dynamic approach uses run-time information on network congestion for selection of routing algorithm. Experimental results show that our proposal improves throughput for real applications up to 37.49%.

They key conclusion of this thesis is that improvement in performance of a computing system needs multifaceted approach i.e., improving the performance of memory and communication subsystem at the same time. The reduction in performance gap between processors and memories requires not only integration of improved memory technologies in system but also software/compiler support. We also conclude that switching routing algorithm with changing traffic pattern of an application leads to improvement of NoC throughput.

Keywords: Memory bound stalls, compiler optimization, execution time, computer vision, non volatile memory, relational DBMS, storage engine, network-on-chip, routing algorithm, throughput.

ÖZET

İŞLEMCİ-BELLEK PERFORMANS FARKINI AZALTMAK VE YONGA ÜSTÜ AĞ VERİMİNİ ARTIRMAK

Naveed Ul Mustafa
Bilgisayar Mühendisliği, Doktora
Tez Danışmanı: Özcan Öztürk
Şubat 2019

Bilişim sistemlerinin performansı son yıllarda, özellikle transistör boyutunun azalması ve saat hızının artması nedeniyle muazzam bir gelişme göstermiştir. Tek bir yonganın üzerine yerleştirilmiş milyarlarca transistör ve yüksek saat hızında anahtarlama, yonganın aşırı ısınmasına neden olmaktadır. Isı dağılımını artırmadan performans iyileştirme gereksinimi, çoklu çekirdeklerin ve/veya belleklerin yonga üzerindeki bir ağ üzerinden iletişim kurduğu çoklu çekirdek tasarımlarının kullanılmasına yol açmıştır. Ne yazık ki, bellek performansı, işlemci performansı ile aynı hızda gelişememiştir ve bu nedenle bellekler bir performans darboğazı haline gelmiştir. Diğer yandan, gerçek uygulamalarda değişen trafik düzenleri, bir rotalama algoritması tarafından gözetilen ağ verimliliğini kısıtlamaktadır.

Bu tezde, işlemci-bellek performans farkı sorununu iki şekilde ele alıyoruz: Birincisi, gelişmiş ve yeni geliştirilen bellek teknolojilerini bir bilgisayar sisteminin bellek hiyerarşisinde birleştiriyoruz. İkincisi, yürütme platformunu gerekli mimari özelliklerle donatıyor ve derleyicinin bellek erişim talimatlarını paralel hale getirmesini sağlıyoruz. Ayrıca, bir Yonga üzeri Ağ (YüA)'nın rotalama algoritmasını, bir uygulamanın değişen trafik düzenine göre değiştiren bir seçim yöntemi önererek, ağ verimliliğinin kısıtlanması sorununu da ele alıyoruz.

Yeni gelişen kalıcı bellek (Non-volatile memory - NVM) cihazlarının bir bilgisayar sisteminin bellek hiyerarşisinde, veri tabanı yönetim sistemleri (DBMS) bağlamında entegrasyonunu sunuyoruz. Bu amaçla, bir DBMS'nin depolama motorunda (SE), DBMS'nin fonksiyonlarının doğru çalışmasını etkilemeden, disk arayüzlerini atlayarak verilere hızlı erişim sağlayan değişiklikler öneriyoruz. Bir uygulama çalışması olarak, PostgreSQL'in SE'sini değiştiriyoruz ve bu değişikliğin gereksinimlerini ve zorlukları detaylandırıyoruz. Önerdiğimiz yaklaşımı kapsamlı bir emülasyon platformu kullanarak değerlendiriyoruz. Sonuçlar, disk ve NVM deposuyla karşılaştırıldığında, değiştirilmiş SE'mizin sorgu sürelerini sırasıyla %19 ve %4'lük bir ortalama azalışla, %45 ve %13'e kadar azalttığını göstermektedir. Bu sonuçların detaylı analizi, değiştirilmiş SE'mizin veri hazırlığı sorununa tâbi olduğunu göstermektedir. Bu sorunu çözmek için, basit bir uygulama programı arayüzü (API) yoluyla NVM donanımından belleğe önceden veri almak için yardımcı iş parçacıkları kullanan genel amaçlı bir kütüphane geliştirdik. Bu kütüphanemiz ile, değiştirilmiş SE için disk ve NVM depolamaya kıyasla sorgu sürelerinin sırasıyla ortalama %23 ve %8'lik bir düşüşle, %54 ve %17'ye kadar azaltılabildiği görülmüştür.

İşlemci bellek performansı farkını azaltmak üzere ikinci bir yol olarak, belleğe bağlı duraklamaların azaltılmasını amaçlayan bir derleyici optimizasyonu öneriyoruz. Önerilen optimizasyon, bellek referanslarının sınıflandırılması yoluyla etkili bir sıralama oluşturmaktadır ve iki adımdan oluşmaktadır: ilginlik analizi ve ilginlik duyarlı sıralama. İlginlik analizi için iki farklı yaklaşımı öneriyoruz; kaynak kod ek açıklaması ve otomatik analiz. Deneysel sonuçlarımız, ek açıklama tabanlı yaklaşımın uygulanmasının, bellek erişimi yoğun bir programda duraklama döngülerini %67,44 azaltarak çalışma süresinde %25,61 kadar iyileşme sağladığını göstermektedir. Ayrıca 11 farklı görüntü işleme değerlendirme deneyi ile otomatik analiz yaklaşımımızı değerlendirdik. Deneysel sonuçlar, otomatik analizin duraklama döngülerini ortalama %69,83 oranında azalttığını göstermektedir. Tüm deneylerde hem hesaplama hem de bellek yoğun işlemler olduğu için, çalışma süresinde ortalama %5.79 iyileşme olmakla birlikte bu oran %30'a kadar çıkmaktadır.

Ağ verimliliğini arttırmak için, rotalama algoritmasını değişen trafik düzenine göre değiştiren bir seçim yöntemi öneriyoruz. İki seçim stratejisi kullanıyoruz: statik ve dinamik seçim. Statik seçim devre dışı bırakıldığında, dinamik yaklaşım, rotalama algoritmasının seçimi için ağ tıkanıklığına ilişkin koşum zamanı bilgilerini kullanmaktadır. Deneysel sonuçlar, yöntemimizin gerçek uygulamalar için %37,49'a kadar verimliliği artırdığını göstermektedir.

Bu tezin temel sonucu, bir bilgisayar sisteminin performansında iyileşme elde etmek için çok yönlü bir yaklaşıma ihtiyaç duyulduğudur, yani aynı anda hem bellek, hem de iletişim alt sisteminin performansının iyileştirilmesi gerektiğidir. İşlemciler ve bellekler arasındaki performans farkının azaltılması için yalnızca gelişmiş bellek teknolojilerinin sisteme entegrasyonu değil, aynı zamanda yazılım/derleyici desteği de gerekmektedir. Ayrıca, bir uygulamanın değişen trafik düzenine göre rotalama algoritmasının değiştirilmesinin, YüA veriminin artmasını sağladığı sonucuna varılmıştır.

Anahtar sözcükler: Belleğe bağlı duraklama, derleyici optimizasyonu, yürütme süresi, bilgisayar görüntüleme, kalıcı bellek, ilişkisel veritabanı, depolama motoru, yonga üstü ağ, yönlendirme algoritması, verim.

I dedicate this work to my MOTHER & FATHER who always supported me at every stage of my life. I can never payback their love and affection.

شکریہ امی جی، شکریہ ابو جی

Acknowledgement

My six and a half year journey of PhD studies at Bilkent University was a pleasant learning experience. It helped me to deepen my knowledge of computer science and gain research experience. It would have not been possible to write this PhD thesis without support of many people. Therefore, I would like to avail this opportunity to pay my deep gratitude to them. First and foremost, I am grateful to Allah Almighty who gave me the opportunity for PhD studies, helped me stay strong and gave me the consistency needed for completion of PhD.

I would like to extend my sincere thanks to my supervisor, Dr. Özcan Özturk, for his excellent supervision and all necessary support needed during PhD studies. Whether it was about adopting a specific strategy for solving a research problem or selection of a simulation platform for evaluation of a proposal, discussions with him and his feedback was always very helpful and time saving. I thank Özcan for enabling me to have collaboration with researchers at Barcelona Supercomputing Center (BSC) and introducing me to HiPEAC community leading to another collaborative research project with Movidius-Intel.

I am also very thankful to Dr. M. Mustafa Özdal who gave me an opportunity to work on Graph Processor project. Although still in progress, the project introduced me to area of graph processors, their challenges and simulation platforms to model potential solution to those challenges.

I am grateful to the members of my thesis committee: Dr. Buğra Gedik, Dr. M. Mustafa Özdal from Bilkent University, Dr. Suleyman Tosun and Dr. Kayhan İmre from Hacettepe University for their valuable comments and discussions. I also show gratitude to my colleagues at Özyeğin University, Dr. Hasan Sözer and Dr. Kübra Kalkan Çakmakçı, for Turkish translation of the abstract.

I would also like to thank Martin J.O'Riordan and Stephen Rogers from Movidius-Intel, Dr. Adrià Armejach, Dr. Adrián Cristal and Dr. Osman Sabri Ünsal from BSC for collaboration and support needed for implementation of research ideas reported in this thesis. I would like to acknowledge Dr. Smail Niar from Université Polytechnique Hauts-De-France (UPHF) for his supervision and helpful feedback in a collaborative research on network-on-chip. He was more than only a research supervisor as his talks were a source of encouragement in facing ups and downs of life during my PhD studies.

My lab mates were also an important part of PhD studies as they contributed towards this thesis in different ways, sometime providing their feedback on improving the draft of a research article, helping in learning a tool, listening to problems of life and learning the Turkish language. I would like to thank Hamzeh Ahangari, Can Fahrettin Koyuncu, Şerif Yeşil, Mohammad Reza Soltaniyeh, Simge Gökçe, Tolga and Gülden for being my wonderful labmates.

I would also like to thank my friends without whom life at Bilkent would have been so boring and dull. Thanks to Ali Haider, Umar Raza, Muhammad Waqas Akbar, Muhammad Sabih Iqbal, Naveed Mehmood, Talha Masood, Abdur Rahman Dandia, Salahudin Zafar, Ali Shiraz, Asad Ali, Abdullah and Zulfiqar for filling my years at Bilkent with colors and fun.

The support I got in form of encouragement, prayers and love from my parents, Muhammad Zawar and Khadim Noor, is unforgettable as they were always there when I needed them. It is due to their prayers that I have successfully fulfilled requirements for PhD. I would also like to thank my beautiful and beloved daughter Eman Azad, my sisters Azra, Saeeda, Hameedah, Misbah, Zeenat and my brother Zia Ul Mustafa Danish for providing me the warmth of the family.

I would like to express my deep thanks to Merve Naveed Ul Mustafa for being a source of support and strength in my last three years at Bilkent, first as my fiancée and then as my wife.

Lastly, I express a deep sense of gratitude to Higher Education Commission (HEC) of Pakistan for financially supporting my PhD.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A computer, by definition, is an entity performing computation [1]. Computers are omnipresent and the driving force of the information age we are living in not only because they compute but also because they store data and perform communication [2]. Like humans, computers are also on a journey of progress from the start of their history. They are expected to provide an ever increasing higher performance since the announcement of first digital computer, ENIAC, in year 1946 [3, 4].

Performance of a computing system is determined by the individual performance of all three of its major components: processor, memory and a communication architecture. Performance of a processor can be characterized in terms of rate at which it performs a given task [5]. Based on this definition, clock rate of a processor is a commonly used performance metric [6]. As shown in Figure 1.1 (reproduced from [7]), clock rate has increased over the years. Coupled with improving instructions per cycle (IPC), increasing clock rate made it possible for processors to achieve performance improvement [8]. At the same time, shrinking transistor size allowed placement of more number of transistors on a single chip [9] (See Figure 1.1), making it possible to perform more logic.

However, billions of transistors placed on a single chip and operating at a high

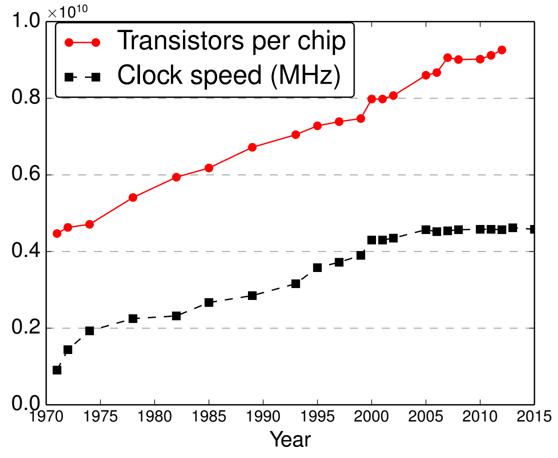


Figure 1.1: Increasing clock speed and number of transistors per chip.

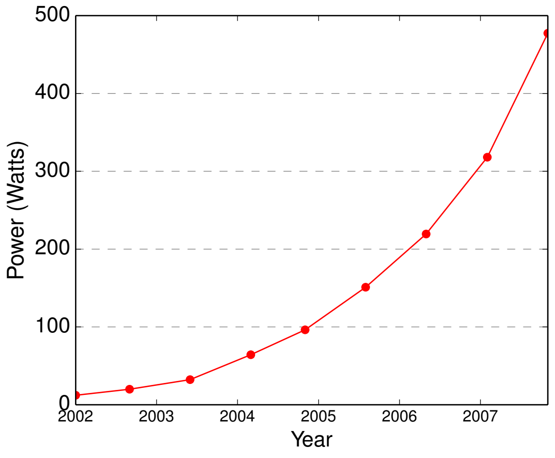


Figure 1.2: Increasing power consumption on a $100mm^2$ die.

frequency result in prohibitively large power consumption, as shown in Figure 1.2 (reproduced from [10]), placing a limit on further increasing the clock rate [11, 12]. As a way to circumvent the power limit and still improve the performance of a processor, researchers proposed to keep frequency increase at a minimum and use multiple simpler cores on a single chip [13, 14].

Multi-core processors exhibit performance gain over a comparable single core processor [15], and hence widely adopted in domains such as image processing [16, 17], embedded systems [18, 19, 20], big data analytics [21], signal processing [22, 23] , and computer vision [24, 25] among others. The wide scale adoption of multi-core platforms is reflected by growing number of cores on a single chip such as intel's Xeon processors, shown in Figure 1.3.

A multi-core design necessitates an on-chip communication architecture to connect multiple cores with each other and/or with memories. Traditionally, point-to-point (P2P) and bus communication architectures have been used for on-chip communication. However, P2P architecture suffers from lack of scalability in terms of complexity and cost (Figure 1.4(a)). On the other hand, bus architecture is simpler than P2P communication (Figure 1.4(b)) but it also lacks scalability and suffers from limited bandwidth [26]. As a result, P2P or bus architectures can not meet the communication needs of multi/many-core era, making

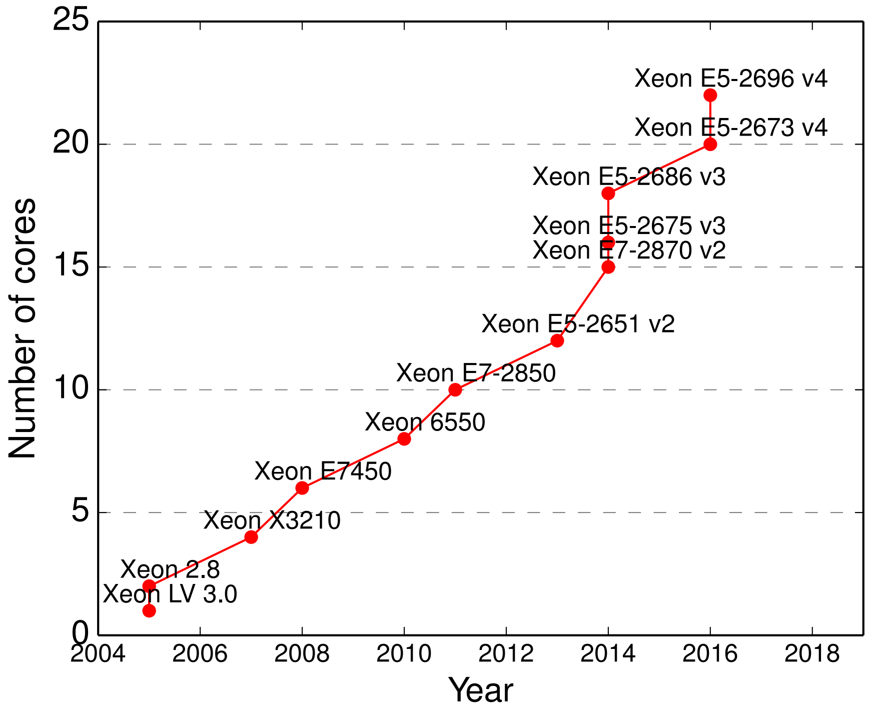
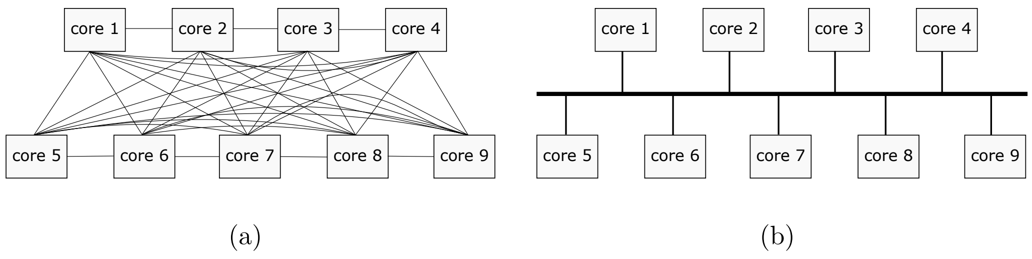


Figure 1.3: Increasing number of cores in Intel's Xeon processors.

the communication a performance bottleneck [27].

Network-on-chip (NoC) has emerged as a viable alternative, serving as communication backbone for many-core based system-on-chip (SoC) designs [28]. An NoC connects processing and storage resources via a scalable network [26] (Figure 1.4(c)) and promises efficient communication between multiple cores, caches, and memory controllers [29].

Processors retrieve data from repositories known as memories. Therefore, performance of computing systems is determined not only by processors and communication architecture but also by memory subsystem. Unfortunately, performance of memory devices has not improved at the same rate as that of processors. There is an increasing performance gap between processors and memories as shown in Figure 1.5 (reproduced from [30]). As a result, for many applications, not processors but the limited bandwidth and long access latency of memory proves to be a serious performance bottleneck [31].



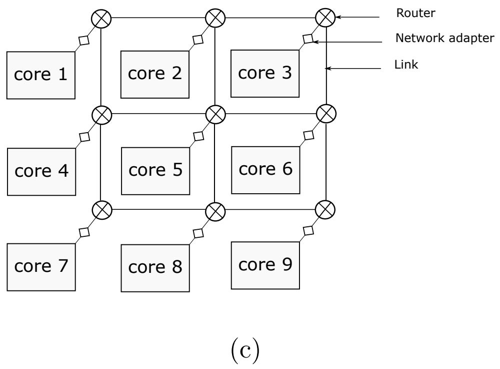


Figure 1.4: Nine cores connected through (a) a P2P (b) a bus architecture (c) and a 3x3 mesh, an example of NoC.

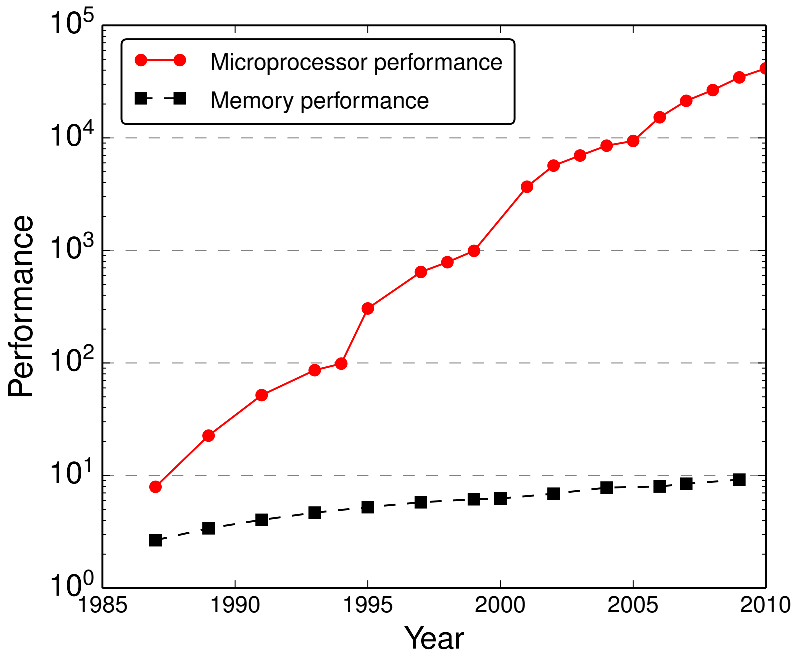


Figure 1.5: Increasing performance (speed) gap between microprocessors and memory.

1.1 Research Problem

Execution time of an application can be divided into two broad categories: commit cycles and stall cycles. A clock cycle is categorized as a commit cycle if at least one instruction is retired during the cycle; otherwise, it is categorized as a stall cycle. Various reasons such as unavailability of functional units, bad branch prediction, or data dependencies result in stall cycles. The unavailability of data required for instruction execution incurs extra clock cycles. Such cycles are termed as memory bound stalls (MBS).

In order to highlight performance bottlenecks of a computing system, we analyze the execution time of a set of compute and memory-intensive benchmarks selected from computer vision domain. Benchmarks perform basic image-processing operations such as image addition and subtraction [32], box filtering [33], convolution [34], sum of absolute difference [35], white balancing operation [36], histogram generation, and similarity measurement between pixels of two input images [37]. Appendix A provides the critical part of the source code for benchmarks.

We use Intel's VTune performance analyzer [38] to breakdown the execution time of benchmarks into commit cycles (CC), bad speculation stalls (BSS), MBS, core bound stalls (CBS), and front-end bound stalls (FEBS). MBS originate from performance gap between processors and memories. Number of MBS depends on level of memory hierarchy holding the required data. Lower the data resides in memory hierarchy, higher is the number of MBS. As shown in Figure 1.6, on average, MBS make almost 33% of the total execution time. In other words, 33% of an application's execution time is spent waiting for data arrival from memory subsystem. It suggests that **performance gap between processors and memory subsystem is a performance bottleneck**.

In a multi-core platform, data transportation between cores and/or memories is performed by an NoC. A routing algorithm determines path of data transport [39] and hence significantly impacts the NoC latency and throughput [40]. Many

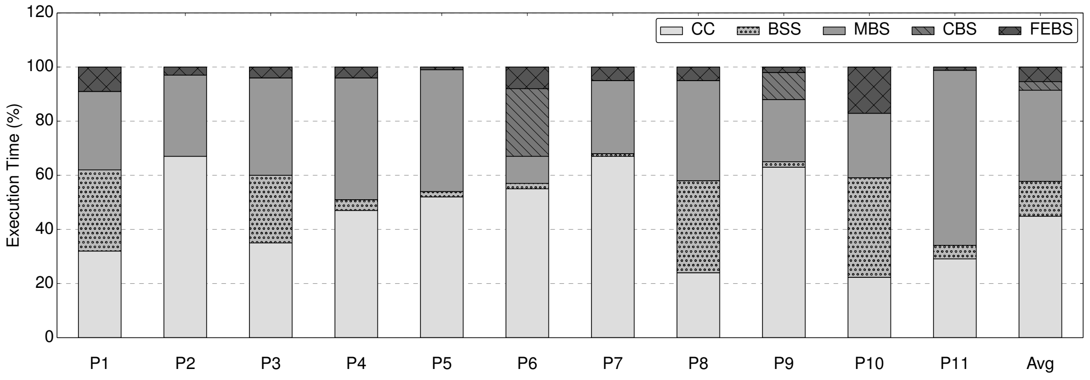


Figure 1.6: Execution time breakdown for benchmarks: P1 = Subtraction of two images, P2 = Addition of four images, P3 = Addition of two images, P4 = Addition of two images based on a mask input, P5 = Box filtering using 5x5 mask, P6 = Addition of two scaled images, P7 = Convolution using 3x3 mask, P8 = Sum of absolute difference using a 5x5 window, P9 = White balancing operation, P10=Histogram generation, P11=Similarity measurement between pixels of two images.

algorithms have been proposed for on-chip routing [41, 42, 43, 44].

Performance of a routing algorithm is affected by traffic pattern which represents the distribution of messages in an NoC [45]. Traffic pattern of a real application is determined by communication links between its tasks mapped on different cores. A source task may have communication links to one or multiple destination tasks i.e. spatial distribution of traffic. Also, tasks may inject data at different time points and at different rates (i.e temporal distribution of traffic). As a result, traffic patterns of real applications are non-uniform and may vary over time.

Deterministic algorithms, such as XY [41], always route data along the same path for a given pair of source and destination. Since traffic pattern of real applications is non-uniform, XY outing is not a good fit for them. Partially adaptive algorithms, such as west-first and north-last [42], allow multiple paths for a source and destination pair. However, performance of such algorithms is affected by traffic pattern. For example, if most of traffic flows leftward in a particular interval of the application's execution period then west-first routing algorithm reduces

Table 1.1: Comparison of different NVM technology with SSD and HDD.

Fetaure	**HDD**	**SSD**	**PCM**	**PRAM**	**MRAM**
Read latency	10 ms	25 μs	50 ns	100 ns	20 ns
Write latency	10 ms	300 μs	150 ns	100 ns	20 ns
Volatile	No	No	No	No	No
Addressability	Block	Block	Byte	Byte	Byte

throughput in that interval. In other words, **applying a single routing algorithm for whole execution period of a real application (with changing traffic pattern) becomes a bottleneck for throughput improvement**.

1.2 Motivation

As an effort to reduce performance gap between processors and memories (shown in Figure 1.5), computing systems use a memory hierarchy organized into several levels and take the advantage of principle of locality in accessing data [46]. A storage device, such as hard disk drive (HDD) or solid state disk (SSD), sits at the lowest level of this hierarchy offering features of non-volatility, highest capacity, lowest cost but longest access latency.

Non volatile memory (NVM) is an emerging storage class technology. Like disk storage, it provides feature of non-volatility (or persistency) but with much lower access latency than SSD/HDD technologies, as shown in Table 1.1. It also supports data access at byte-level granularity [47]. Due to its promising features, NVM has been investigated for use in computing systems as an on-chip cache[48], main memory and storage device [47, 49].

We believe that features of NVM can be exploited to reduce performance gap between processors and memory subsystem of an I/O-intensive computing systems. An example of I/O-intensive computing system is read dominant decision support systems (DSS). They are computer technology solutions used to facilitate complex decision making [50] and convert business information into tangible results [51] to help executives take knowledge-based decisions. DSS frequently read huge data sets from system storage.

A database management system (DBMS) is a necessary component of DSS. In a traditional DBMS, HDD is used to provide means of durable storage. However, reads and write operations to an HDD are very expensive. Performance of memory subsystem (including storage device) can be improved by avoiding expensive I/O operations through replacement of HDD/SSD with an NVM storage device.

Another way to reduce performance gap between processors and memories is to reduce MBS by parallelizing multiple memory accesses. The parallelization requires architectural support not only at the level of memory subsystem (for example, multi-port and multi-bank memories) but also at core-level (for example, multiple load-store units). However, architectural innovations without proper software support are not sufficient. **Compilers need to generate efficient instruction schedule by considering architectural features for reduction of MBS**.

Performance of a computing system is also affected by its communication architecture. For example, higher NoC latency or lower throughput increases the travel time of data from memories to processors and hence result in higher number of MBS. A routing algorithm significantly impacts NoC throughput. At the same time, performance of an algorithm is affected by traffic pattern. To highlight effect of an application's changing traffic pattern on network throughput delivered by a routing algorithm, we analyze performance of west-first and north-last algorithms. We simulate a real application i.e., H.264 video decoder with a resolution of 720p.

Variations in the network throughput delivered by west-first and north-last routing algorithms, as shown in Figure 1.7, can be interpreted in two complementary ways. First, varying traffic pattern of an application affects the throughput of routing algorithms. Second, **network-throughput can be improved by switching routing scheme to an appropriate algorithm with changing**

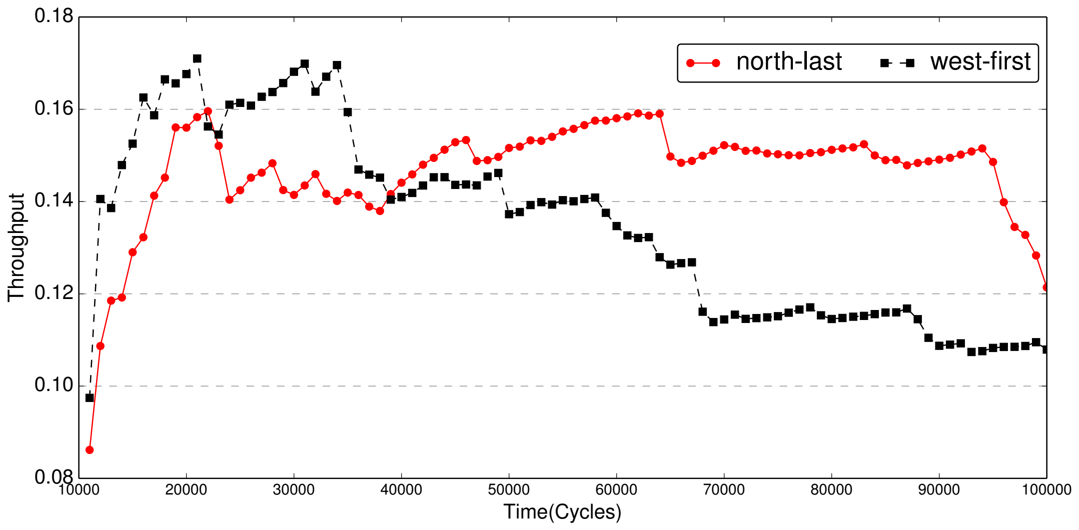


Figure 1.7: Network throughput for H.264 video decoder with a resolution of 720p.

traffic pattern of an application. For example, network throughput in case of H.264 video decoder application can be improved if there is a mechanism to apply west-first routing in simulation interval of 10K to 48K cycles and than switch to north-last routing in simulation interval of 48K to 100K cycles.

1.3 Thesis Statement

Our thesis statement is:

1. Performance gap between processors and memories of a computing system can be reduced by integrating improved and newly developed memory technologies in its memory hierarchy.
2. MBS can be reduced by parallelizing memory accesses through an optimized architecture-aware compiler.
3. NoC throughput can be improved by devising a mechanism for selection of

appropriate routing algorithms with changing traffic pattern of real applications.

1.4 Contributions

In this thesis we mainly contribute by proposing solutions to improve performance of memory subsystem and NoC. More specifically, this thesis makes following main contributions.

1. We discuss and provide insights on different available options when including NVM into the memory hierarchy of computing systems for reducing performance gap between processor and memory subsystem. We address the problem in the context of DBMS. We focus on investigating the necessary changes and challenges when modifying an existing, well-tested, widely used, and robust traditional DBMS to benefit from NVM features. As a case study, we select PostgreSQL which is ranked as the 4th most popular DBMS [52].

   We propose modification in the storage engine (SE) of a DBMS aiming at providing fast access to data by bypassing the slow disk interfaces while maintaining all the functionalities of a robust DBMS such as PostgreSQL. We evaluate our proposed modification in SE of PostgreSQL using a comprehensive emulation platform. We identify and quantify performance bottlenecks that appear when employing NVM hardware. To further improve the performance of our NVM-enabled SEs, we design and implement a general purpose data prefetching library based on helper threads that tackles the identified performance bottlenecks.

   We evaluate our modified SE by running TPC-H benchmark [53]. Evaluation results indicate that modified SE (without usage of prefetching library) reduces query execution time by up to 45% and 13% when compared to disk and NVM storage, with average reductions of 19% and 4%, respectively. Modified SE coupled with data prefetching library further improves query execution time when compared to disk and NVM storage by up to 54% and 17%, with average reductions of 23% and 8%, respectively. This work is presented in SAMOS 2016 conference and published in its proceedings [54].

2. We propose a compiler optimization aiming at reduction of MBS through classification of memory references. The optimization consists of two steps: affinity analysis and afnity-aware instruction scheduling (AAIS). While affinity analysis predicts the physical memory location for each memory object in the application's source code, AAIS generates a stall-saving instruction schedule based on the results of affinity analysis step. We propose two different affinity analysis approaches, namely, source code annotation and automated analysis. We present implementation of the proposed optimization on LLVM compiler infrastructure.

   We evaluate annotation-based approach on a memory-intensive program achieving a reduction in stall cycles by 67.44%, leading to 25.61% improvement in execution time. We also use 11 different image-processing benchmarks for evaluation of automated analysis approach. Our experimental results show that proposed optimization reduces stall cycles, on average, by 69.83%. As all benchmarks used for evaluation of automated analysis approach are both compute and memory-intensive, we achieve improvements in execution time by up to 30%, with a modest average of 5.79%. This work is published in JRTIP [55].

3. We propose a selection scheme to improve throughput of an NoC by selecting appropriate routing algorithms with changes in the traffic pattern of an application. We aim at selecting a routing algorithm which delivers higher throughput for traffic pattern constituted by active communication flows in each interval, and hence improve the overall network throughput for execution period. We propose two different approaches for algorithm selection: static selection and dynamic selection. We evaluate our proposal by executing different benchmarks on NoCs of different sizes. Experimental results show that our proposal improves throughput by up to 37.49%. This work is published in proceedings of RAPIDO 2016 workshop [56].

1.5 Outline

Thesis is organized into 7 Chapters. Chapter 2 provides necessary background on proposed techniques for reducing processor-memory performance gap and improving NoC throughput. It also presents a brief survey of prior related work. Chapter 3 elaborates on integration of NVM in memory hierarchy of a DBMS. Chapter 4 explains the compiler optimization for reduction of MBS. Chapter 5 presents the selection scheme for routing algorithms in order to improve network throughput. Chapter 6 provides the detailed experimental evaluation of our proposed techniques. Chapter 7 concludes the thesis while Appendix A provides supplementary data.

Chapter 2

Background

This chapter is organized into three parts. In the first part, consisting of Section 2.1 to 2.3, we describe DBMS from memory hierarchy perspective, explain properties of NVM technologies, highlight the implications these features might have in the design of a DBMS and describe prior related work on usage of NVM in DBMS design.

In the second part, consisting of Section 2.4 to 2.6, we present a brief introduction of computer vision applications and the architecture of execution platform used for evaluation of the proposed compiler optimization (detailed in Chapter 4). We also present a brief survey of prior related work on reduction of MBS and compare it with the proposed compiler optimization.

In the third part, consisting of Section 2.7 and 2.7.1, we discuss NoC routing algorithms, traffic patterns and their effect on algorithm performance along with a brief survey of prior work on reconfiguring NoC routing algorithm.

2.1 Memory Hierarchy of a DBMS

The traditional design of a DBMS assumes a memory hierarchy where datasets are stored in disks. Disks are a cheap and non-volatile storage medium suitable for storing large datasets. However, they are extremely slow for data retrieval. To hide their high data-access latency, DRAM is used as an intermediate storage between disks and the processing units.

DRAM is orders of magnitude faster than a disk, and with increasing DRAM chip densities and decreasing memory prices, relational in-memory DBMSs have become increasingly popular [57, 58, 59, 60, 61, 62]. Significant components of in-memory DBMSs, like index structures [63, 64], recovery mechanisms for system failure [65, 66], and commit processing [67] are tailored towards the usage of main memory as primary storage. An in-memory DBMS assumes that all data fits in main memory. However, for some applications, the database size can grow larger than DRAM capacity [68]. At the same time, inherent physical limitations related to leakage current and voltage scaling limit the further scaling of DRAM [69, 70]. The capacity problem can be resolved by distributing the database across multiple machines but at the cost of performance degradation [68]. Furthermore, due to the volatile nature of DRAM, in-memory DBMSs still use a large pool of disks to provide a form of persistent storage for critical or non redundant data [57, 71, 72, 73].

NVM is an emerging storage class technology that features persistency as well as significantly faster access latencies than hard disks, with read latencies on the same order of magnitude as DRAM [74]. It also offers byte-addressability like DRAM and higher density [75, 76]. Prominent NVM technologies are PC-RAM [1] [77], STT-RAM [2] [70], and R-RAM [3] [78]. With read latency close to that of DRAM, especially in case of PC-RAM and R-RAM [79, 80], NVM technologies are a good candidate to improve the performance of decision support systems (DSS), which are dominated by read-only queries on vast datasets [81]. DSS are

[1]PC-RAM: Phase Change Random Access Memory

[2]STT-RAM: Spin Transfer Torque Random Access Memory

[3]R-RAM: Resistive Random Access Memory

computer technology solutions that can be used to facilitate complex decision making [50]. They convert business information into tangible results [51] to help executives take knowledge-based decisions.

As a DBMS is a necessary component of DSS, its design should take into account the characteristics of NVM (described in Section 2.2) to benefit from its features. Simple ports of a traditional DBMS - designed to use disks as the primary storage medium - to NVM will show improvement due to the lower access latencies of NVM. However, adapting a DBMS to fit NVM characteristics can offer a number of benefits beyond lower access latencies.

This adaptation requires modifications in the storage engine (explained in Section 3.3) as well as other components of a DBMS [74] in order to take advantage of NVM features. As explained in Section 6.1.4, a SE modified for NVM storage suffers from data readiness problem. To solve the issue, we also develop a general purpose library (explained in Section 3.7) to prefetch data seamlessly in a timely fashion.

2.2 Characteristics of NVM

Data access latency: Read latencies for NVM technologies will certainly be significantly lower than those of conventional disks. However, since NVM devices are still under development, sources quote varying read latencies. For example, the read latency for STT-RAM ranges from 1 to 20ns, and PC-RAM is expected to be around 50ns [79, 82, 83]. Nonetheless, read latency of some NVM technologies is expected to be similar to that of DRAM [47, 79, 80, 82, 84, 85, 86], which is typically around 60ns.

PC-RAM and R-RAM are reported to have a higher write latency compared to DRAM, but STT-RAM also outperforms DRAM in this regard [79, 82]. However, the write latency is typically not in the critical path, since it can be tolerated by using buffers [75].

Density: NVM technologies provide higher densities than DRAM, which makes them a good candidate to be used as main memory as well as primary storage, particularly in embedded systems [87]. For example, PC-RAM provides 2 to 4 times higher density as compared to DRAM [75]. Future NVMs are expected to have higher capacity and better scalability than DRAM [49, 88, 89, 90]

Endurance: The maximum number of writes a memory cell can withstand is lower for most NVM technologies when compared to DRAM [75, 91]. Specifically, PC-RAM, R-RAM, and STT-RAM have projected endurances of 10^{10}, 10^{8}, and 10^{15} respectively; as compared to 10^{16} for DRAM [79]. On the other hand, NVMs exhibit higher endurance than flash memory technologies [82].

Energy consumption: Since NVM does not need a refresh cycle to maintain data states in memory cells like a DRAM, they are more energy efficient. A main memory designed using PC-RAM technology consumes significantly lower per access write energy as compared to DRAM [91]. Other NVM technologies also have similar lower energy consumption per bit when compared to DRAM [79, 83].

In addition to the features listed above, NVM technologies also provide byte-addressability like DRAM and persistency like disks. Due to these features, NVMs are starting to appear in embedded and energy-critical devices and are expected to play a major role in future computing systems. Companies like Intel and Micron have launched the 3D XPoint memory technology, which features non-volatility [92]. Intel has also introduced new instructions to support the usage of persistent memory at the instruction set architecture (ISA) level [93]. In the following section, we present a brief survey of prior work on usage of NVM in DBMS design and its integration in memory hierarchy.

2.3 Prior Work on Usage of NVM in DBMS Design

Previous work on usage of NVM in the context of DBMS can be divided into three broad categories: proposals for (i) NVM-aware DBMS designs from scratch, (ii) modification of one or more components of an already existing in-memory DBMS, and (iii) using NVM in a disk-oriented DBMS.

In the first category, Arulraj *et al.* [79] propose usage of a single tier memory hierarchy, i.e., without DRAM, and compare three different storage management architectures using an NVM-only system for a custom designed lightweight DBMS. In [74], Arulraj *et al.* discuss designing a DBMS for NVM and suggest that an NVM-enabled DBMS needs to adapt the logging protocol as well as the in-memory buffer cache in order to achieve significant performance improvements. Peloton [61] is another example of a DBMS designed from scratch for DRAM/NVM storage.

In the second category, Pelley *et al.* [94] explore a two-level hierarchy (i.e., DRAM coupled with NVM) and study different recovery methods, using Shore-MT [95] storage engine. Others have suggested employing NVM only for logging components of a DBMS and not for dataset storage. For example, earlier works implement NVM-Logging in Shore-MT and IBM-SolidDB [96, 97] to reduce the impact of disk I/O on transaction throughput and response time by directly writing log records into an NVM component instead of flushing them to disk. Wang *et al.* [98] demonstrate, by modifying Shore-MT, the use of NVM for distributed logging on multi-core and multi-socket hardware to reduce contention of centralized logging with increasing system load.

While in-memory DBMS have become quite popular, disk-based DBMS still have not lost their importance as indicated in top ten ranking of DBMS by popularity [52]. Therefore, researchers have investigated use of NVM in the context of disk-based DBMS. In this third category, Gao *et al.* [99] use phase changing memory (PCM) to hold buffered updates and transaction logs in order to improve transaction processing performance of a traditional disk-based DBMS (i.e. PostgreSQL 8.4). NVM has also been used to replace a disk-located double write buffer (DWB) in MySQL/InnoDB by a log-structured buffer implemented in NVM, resulting in higher transaction throughput [100].

The work presented in this thesis belongs to the third category. It focuses on usage of NVM as a replacement of disk storage in a traditional DBMS and explains necessary changes in the storage engine for such a replacement. Our contribution is an NVM-aware storage engine that is complementary to and can be applied along with PCM-logging [99] and NVM-buffering [100] on a traditional disk-based DBMS.

Helper threads have been used for parallel speedup of legacy applications. Researchers have used programmer-constructed helper threads [101] as well as compiler algorithms [102] for automated extraction of helper threads to enhance the performance of applications. Use of helper-threads for loosely coupled multiprocessor systems is demonstrated in [103], with focus on efficient thread synchronization for low overhead. Others have used special hardware to construct, spawn, optimize and manage helper threads for dynamic speculative precomputation [104]. Although helper-thread based prefetching is a well-studied technique, we pioneer its use in the context of NVM storage for DBMS in order to resolve the data readiness problem arising from having direct access to NVM-resident data.

2.4 Computer Vision: Applications and Platforms

Computer vision (CV) is a rapidly growing field, mostly devoted to capturing, analysis, modification, and understanding of images [105, 106]. With the arrival of high-resolution cameras in mobile devices, CV applications are becoming more popular [105]. Embedded systems such as wearable devices, drones, robots, and tablets are supposed to support CV applications [107]. Domains that employ CV include surveillance [108, 109] gesture recognition [110], face tracking [111, 112], medical imaging [113, 114], automotive safety [115, 116], and food industry [117, 118, 119], among others.

CV applications are computationally expensive and mostly required to execute in real time [105]. However, embedded platforms are limited on the power budget. There are two architectural solutions to reduce the power consumption and running the CV algorithms faster on embedded systems. One popular approach is to use a multi-core platform. In general, two smaller cores collectively occupying the same area and consuming the same energy as compared to a single large core can provide 70–80% higher performance [120]. The other possible approach is using the dedicated optimized cores to implement the commonly used algorithms. This can be achieved using domain-specific hardware accelerators [105]. Besides employing architectural solutions, it is critical for a compiler to reduce the execution time of applications by taking into account the architectural features of the hardware platform [121].

There have been various efforts to design vision-processing systems targeting CV applications such as [24, 121, 106, 122], among others. One such effort is Myriad 2 platform from Movidius [123]. It is a low-power multi-processor system on chip (MPSoC) that uses an array of very long instruction word (VLIW) processors with vector and single instruction multiple data (SIMD) execution capabilities [124]. Since CV applications are heavy in both computation and memory requirements [125], the platform features a high bandwidth memory subsystem. As we use Myriad 2 as execution platform for evaluation of proposed compiler optimization, we present its architectural details and memory subsystem in the following section.

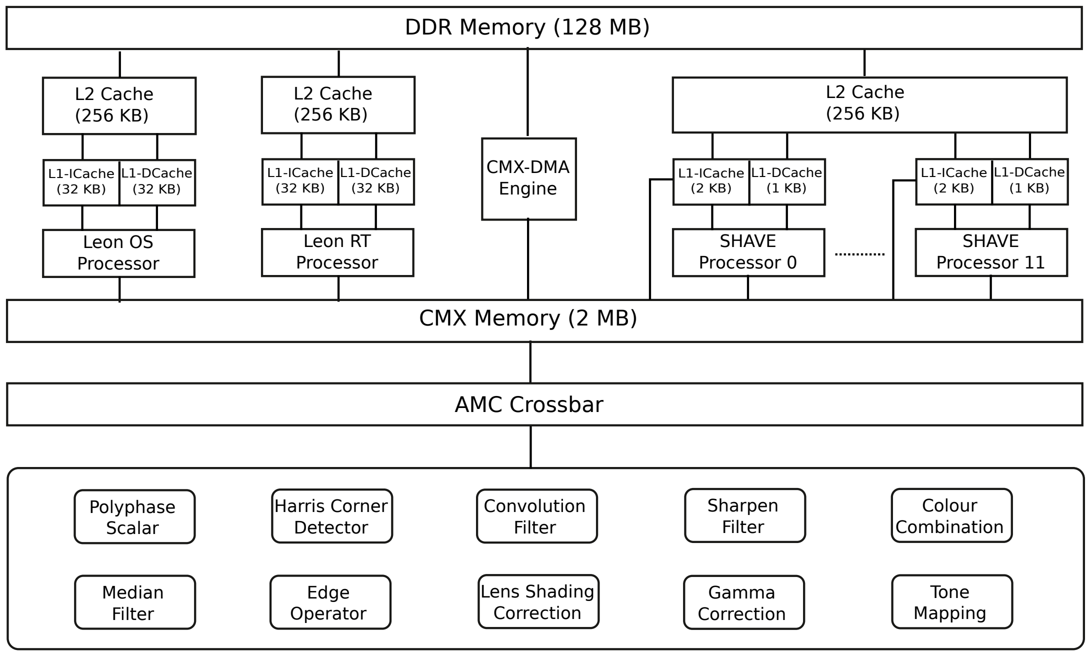


Figure 2.1: Architectural layout of a Myriad 2 platform.

2.5 Myriad 2 Architecture

Figure 2.1, based on [126, 127], shows the architectural layout of a Myriad 2 Platform developed by Movidius Ltd [123]. It is an MPSoC containing multiple heterogeneous processors, hardware accelerators, memories, and external interfaces. Target application domain for the Myriad 2 platform is video filtering and image recognition in embedded systems [127].

Myriad 2 contains 12 streaming hybrid architecture vector engine (SHAVE) and two reduced instruction set computing (RISC) processors. SHAVE processors are the real workhorse of Myriad 2 and are designed to crunch the complex imaging and vision algorithms [126]. The platform offers a 2 MB connection matrix (CMX) memory along with a number of programmable hardware accelerators for vision processing. Accelerators are connected to the CMX memory via a crossbar [107].

SHAVE is a VLIW processor containing a set of functional units which are fed with operands from three different register files [124]. The processor contains

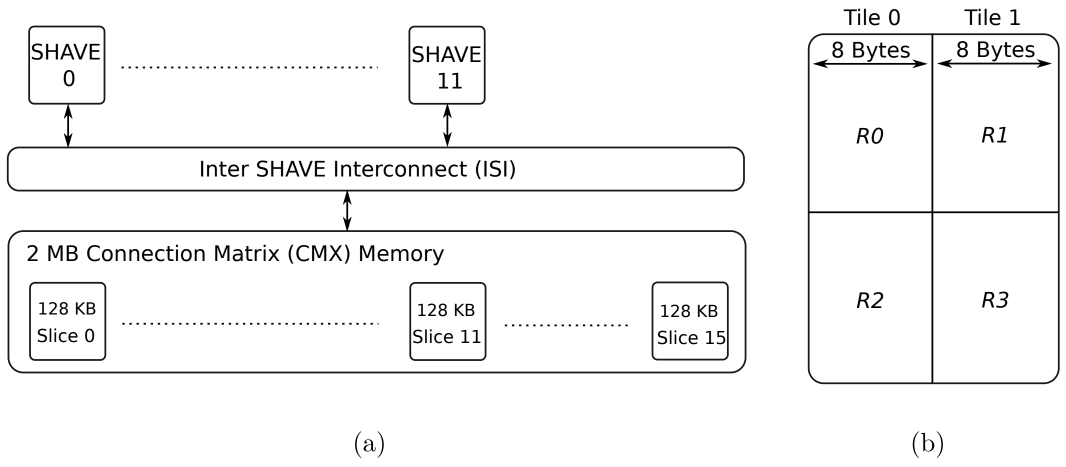


Figure 2.2: Organization of the CMX memory and its interface with SHAVE processors in Myriad 2. Memory is divided into slices (a) and each slice is divided into regions (b).

optimized functional units such as a branch and repeat unit (BRU), a compare and move unit (CMU), arithmetic units, and two load-store units (LSUs). Each SHAVE processor can execute two load-store instructions simultaneously.

2.5.1 CMX Memory

As shown in Figure 2.2(a), the 2 MB CMX memory is divided into 16 different slices, each with a size of 128 KB. A slice can hold both instructions and the data for a program running on a processor. Each of the first twelve slices (i.e., slice 0–slice 11) has an affinity to 1 of 12 SHAVE processors. Since Myriad 2 is a non-uniform memory access (NUMA) platform, it is more efficient in terms of latency and energy consumption for a processor to access its local slice (i.e., slice 0 for SHAVE 0). However, processors can also access any other slice in the CMX memory but with a higher latency. Therefore, placement of data in the local slice of a processor is recommended.

A slice is further divided into four regions, named R0, R1, R2, and R3 in Figure 2.2(b), each with a size of 32 KB. In principle, the architectural design of the CMX memory allows four simultaneous memory accesses in four different regions of a given slice. Each region is a single physical block of random access memory (RAM) with a single chip select and a single set of address and data paths. Therefore, simultaneous memory accesses in the same region are not recommended as they result in stall cycles due to clash among memory ports. Since a SHAVE processor has only two LSUs, only two simultaneous memory accesses are practically possible into a single CMX slice. Simultaneous memory accesses can be performed in any of the two different regions, e.g., R0 and R1 or R0 and R2.

Although Myriad 2 features a high bandwidth memory subsystem, its compiler schedules memory accesses inefficiently. The basic reason is that compiler is unaware of the memory organization. This results in unnecessary memory stalls and hence higher execution time for applications. In Chapter 4, we propose an optimization based on classification of memory references and taking advantage of memory organization to reduce memory bound stalls. In the following section, we present a brief survey of various approaches adopted in previous work for reduction of memory bound stalls.

2.6 Prior Work on Reduction of Memory Bound Stalls

Memory bound stalls cause under-utilization of the compute logic due to memory latency and hence become a major hurdle in improving the execution time of an application [128]. Various approaches have been proposed to reduce memory stalls, such as data mapping in multi-bank environment, using non-uniform memory access (NUMA)-based design and architectural improvements in the memory and compute fabric.

Platforms with multi-bank memory system mitigate the problem by mapping simultaneously requested data on different memory banks. Researchers have presented proposals to implement data mapping as a back-end compiler optimization [129, 130] as well as by analyzing memory access pattern at higher levels [131, 132, 133] for single-processor systems. Other works, such as [134, 135], propose approaches for mapping data of different applications to multiple memory banks in a multi-core environment.

NUMA is commonly used in modern multi-processor systems to avoid the bottleneck of shared memory accesses [136]. It provides asymmetric memory bandwidth and latency characteristics [137]. In other words, cost of accessing data located in remote memory modules is higher than accessing data blocks in local memory modules. Memory affinity is a way to reduce this cost by placing the data in memory modules closer to the processor executing the computation thread [138] and guarantees to improve memory bandwidth and latency [139].

Many researchers have contributed in the context of memory affinity to reduce memory access cost on NUMA platforms. For example, a NUMA API for Linux was proposed in [140] which allows programmers to make memory allocations from a specific node or memory module, in addition to binding threads to specific CPUs. Different algorithms have been proposed to spread application data across different memories of a NUMA platform, such as round-robin, first touch affinity and next-touch affinity [136, 141]. An extension to Linux kernel to add support for the affinity-on-next-touch algorithm is reported in [142].

In this work, we exploit the availability of dual load-store units to processors of a vision-processing system and its NUMA architecture, where memory is divided into multiple slices, each one having an affinity to one of the processors. Unlike the traditional memory affinity approach focusing on the reduction of latency by placing data closer to the computing processor [136, 137, 141, 143], the purpose of our affinity analysis is to reduce the memory bound stalls by taking into account memory organization and hence efficiently scheduling memory accesses.

Another approach to reducing memory stalls, more related to our work, is optimization of the memory subsystem of the execution platform and related architectural components of the compute fabric. Like Myriad 2 platform, Snapdragon 800 [144], MaPU [121], and TI AcceleratorPac [122] use VLIW processors as main execution units [24] combined with RISC cores and other dedicated components. Unlike these systems using unified memory, Myriad 2 uses NUMA architecture enabling multiple cores to access their local memory slices simultaneously and hence make a contribution in reducing memory stalls.

Hexagon DSP on Snapdragon 800 is a VLIW featuring two data units. Each data unit is capable of executing a load, a store or an ALU instruction but unable to pack two memory accesses with one or more ALU instructions in a single cycle. On the other hand, VLIW processors of Myriad 2 are capable of packing two memory accesses with up to two ALU instructions in a single cycle.

MaPU platform contains ten processing cores with unified memory scheme. A core can make up to three memory accesses simultaneously but into different physical memories. Furthermore, a physical memory cannot be accessed by different cores simultaneously. As compared to MaPU, Myriad 2 supports simultaneous accesses to memory at two levels. First, multiple cores can access their local memory slices simultaneously due to NUMA architecture. Second, each core can make up to two simultaneous accesses into its local slice.

As noted by designers of MapU [121], compilers are a major source of the lower performance of execution platforms as they use a simplified model of processor architecture and do not consider detailed architectural features of the platform. Since our proposed compiler optimization is based on comparatively better architectural features of Myriad 2 platform (such as dual load-store units per processor and a high bandwidth memory subsystem), as shown in Table 2.1, it is not only different than Hexagon DSP and MaPU, but has a potential of achieving higher performance.

Table 2.1: Comparison of architectural features of different CV platforms (read DNA as "Details not available").

Platform	**Uses NUMA architecture**	**Support for packing more than two memory instructions with an ALU instruction**	**Support for multiple simultaneous accesses by a single processor to the same physical memory.**
Snapdragon 800 (with Hexagon DSP)	No	No	Yes
MaPU	No	Yes	No
TI AcceleratorPAC	No	DNA	DNA
Myriad 2	Yes	Yes	Yes

2.7 Network-on-Chip Routing

A System on Chip (SoC) consists of multiple Processing Elements (PEs) on a single chip. An NoC is a communication medium among PEs in a SoC [145]. Most current NoC architectures employ a 2D mesh topology [146] where processing elements are arranged in form of rows and columns connected through communication links. Data injected by PEs into mesh, through network interfaces, is routed across mesh through routers. Figure 2.3 shows 16 PEs connected through a 4x4 mesh.

A router implements a routing algorithm to transport data from a source to a destination PE. Performance of a routing algorithm is affected by traffic pattern which is determined by behavior of active communication flows of an application. As an application's tasks, mapped on different PEs, may not communicate always at the same data rate, its traffic pattern may change during execution.

A given routing algorithm may perform well for one traffic pattern but give poor results for others. Therefore, applying a single routing algorithm for whole

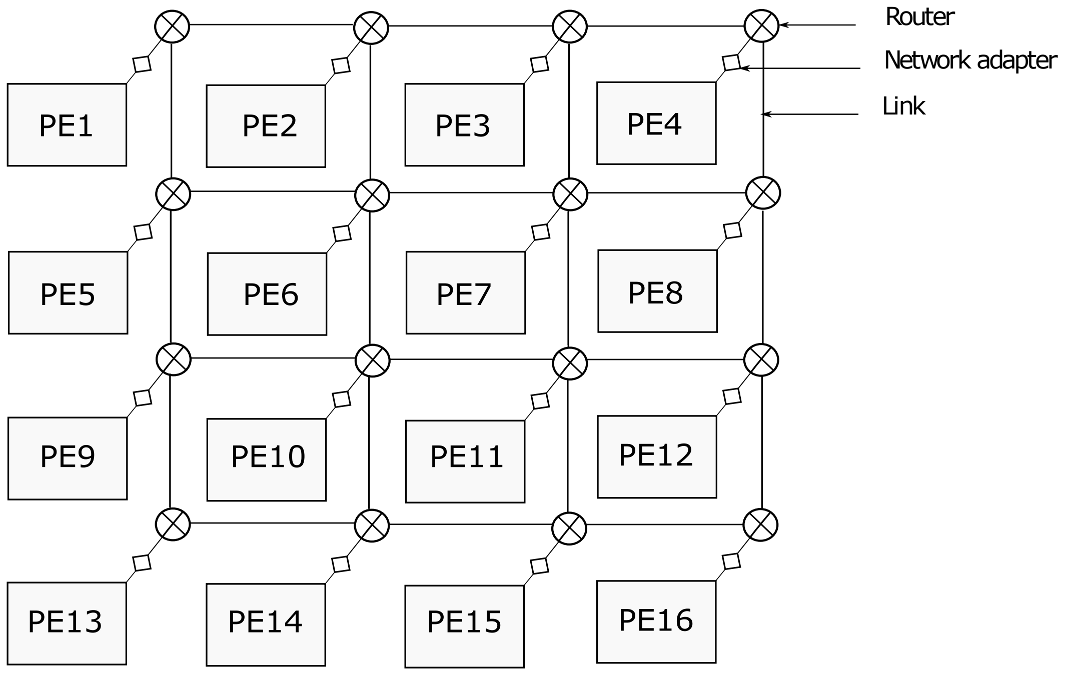


Figure 2.3: 16 PEs connected through a 4x4 mesh NoC.

execution period of an application limits the network throughput. In Chapter 5, we present a selection scheme for selecting an appropriate NoC-routing algorithm with changes in the traffic pattern of an application. In the following subsection, we present a brief survey of prior work on reconfiguring NoC routing algorithms and highlight differences of our approach.

2.7.1 Prior Work on Reconfiguring Noc Routing Algorithm

Many of Dynamically Reconfigurable NoCs (DRNoCs) can be reconfigured in terms of architecture or routing algorithm, for example, DyNoc [147], CoNoChi [148] and DRAFT [149]. In order to take care of the dynamically changing architecture in DyNoC, XY routing algorithm is adapted to deal with obstacles created by the dynamic placement and removal of modules. CoNoChi generates a routing table whenever topology changes and tables are then distributed via the network. In DRAFT, a hierarchal level dependent mask is used by each router to determine if the received flit is to be moved upward [150].

All of the above mentioned DRNOCs reconfigure a particular routing algorithm to make it compatible with the modified network architecture. However, in our approach network architecture is fixed. We use a static mesh network and focus on switching the routing algorithm to take advantage of variations in traffic pattern during execution for improving throughput.

In [44], DyAD routing technique is described which selects between two routing algorithms (a deterministic and an adaptive one) based on the congestion threshold and congestion flags. However, we investigate the possibility of making selections among a deterministic algorithm, such as XY routing, and multiple adaptive algorithms (for example, west-first, negative-first and north-last routing) based on the measurement of congestion level.

Chapter 3

Integrating NVM in Memory Hierarchy of a DBMS

In this chapter, we first discuss the possible memory hierarchy designs when including NVM in a DBMS. We then describe currently available system software to manage NVM. We also explain the high-level modifications necessary in a traditional disk-optimized DBMS in order to take full advantage of NVM hardware. As a case study, we apply these modifications on storage engine (SE) of PostgreSQL.

Our modified SEs of PostgreSQL show performance improvement. However, the performance is limited by data readiness problem. Therefore, in this chapter, we also describe a general purpose library to prefetch data using known techniques like helper threads. In Chapter 6, we evaluate modified SEs with and without usage of library services.

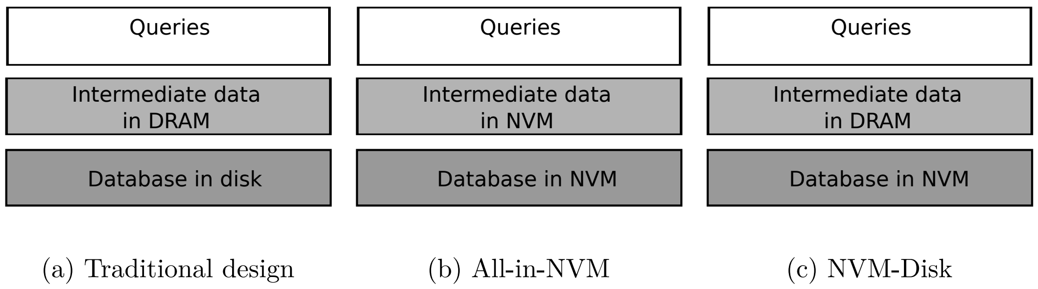


Figure 3.1: NVM placement in the memory hierarchy of a DBMS.

3.1 Memory Hierarchy Designs for an NVM-Based DBMS

With features of byte-addressability, low latency and high capacity, NVM has the potential to replace traditional disks as well as main memory [80]. Figure 3.1 shows different options that might be considered when including NVM into the system. Figure 3.1(a) depicts a traditional approach, where the intermediate state - including logs, data buffers, and partial query state - is stored in DRAM to hide disk latencies for data that is currently in use; while the bulk of the relational data is stored in a disk.

Given the favorable characteristics of NVM over the other technologies, an option might be to replace both DRAM and disk storage using NVM (Figure 3.1(b)). However, such a drastic change would require a complete redesign of current operating systems and application software. In addition, NVM technology is still not mature enough in terms of endurance to be used as a DRAM replacement. Hence, we advocate for a platform that still has a layer of DRAM memory, like [151], where the disk is completely or partially replaced using NVM, as shown in Figure 3.1(c) (NVM-Disk).

Using this approach, we can retain the programmability of current systems by still having a layer of DRAM, thereby exploiting DRAM's fast read and write access latencies for temporary data structures and application code. In addition, it allows the possibility to directly access the bulk of the database relational data by using a file system such as persistent memory file system (PMFS, described in Section 3.2), taking full advantage of NVM technology, which allows the system to leverage NVM's byte-addressability and to avoid API overheads [96] present in current files systems (FSs). Unlike an in-memory DBMS, such a setup does not need large pools of DRAM since temporary data is orders of magnitude smaller than the actual relational database stored in NVM. We believe this is a realistic scenario for future systems integrating NVM, with room for small variations such as NVM alongside DRAM to store persistent temporary data structures, or having traditional disks to store cold data.

3.2 System software for NVM

Using NVM as primary storage necessitates modifications not only in application software but also in system software in order to take advantage of NVM features. A traditional FS accesses the storage through a block layer. If a disk is replaced by NVM without any modifications in the FS, the NVM storage will still be accessed at block level granularity. Hence, we will not be able to take advantage of the byte-addressability feature of NVM.

For this reason, there have been developments in file system support for persistent memory. PMFS is an open-source POSIX compliant FS developed by Intel Research [152, 153]. It offers two key features in order to facilitate usage of NVM.

First, PMFS does not maintain a separate address space for NVM. In other words, both main memory and NVM use the same address space. This implies that there is no need to copy data from NVM to DRAM to make it accessible to an application. A process can directly access file system protected data stored in NVM at byte level granularity.

Second, in a traditional FS stored blocks can be accessed in two ways: (i) file I/O and (ii) memory mapped I/O. PMFS implements file I/O in a similar way to

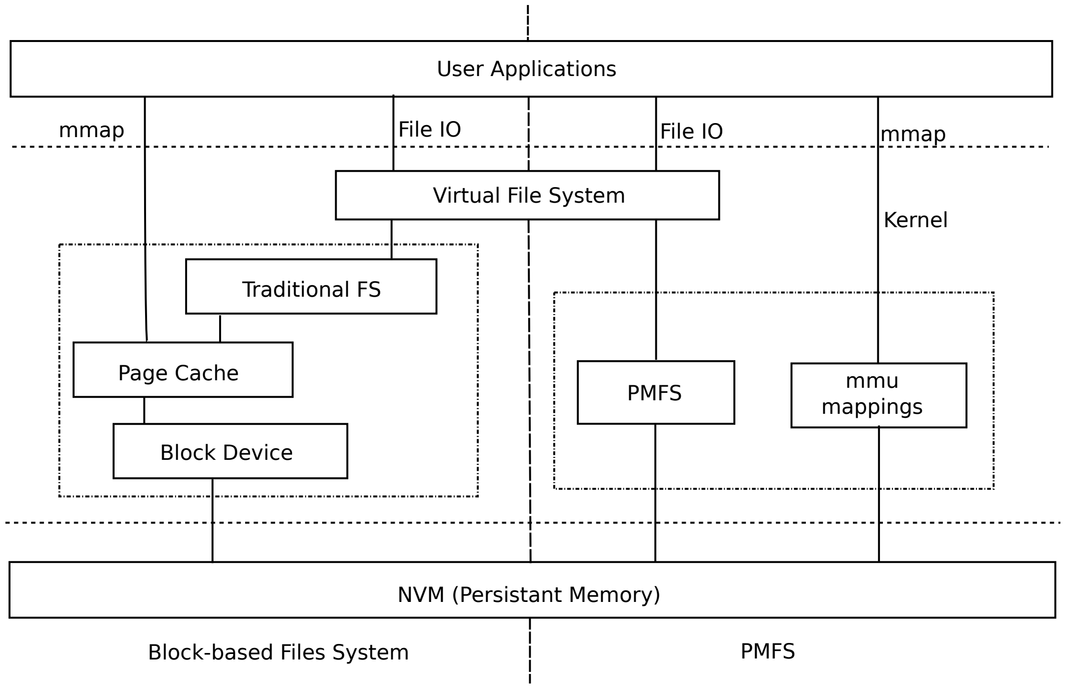


Figure 3.2: Comparison of traditional FS and PMFS. “mmap” refers to the system call for memory mapped I/O operation. “mmu” is the memory management unit responsible for address mappings.

a traditional FS. However, the implementation of memory mapped I/O differs. In a traditional FS, memory mapped I/O would first copy pages to DRAM [152] from where an application can examine those pages. PMFS avoids this copy overhead by mapping NVM pages directly into the address space of a process. Figure 3.2 from [152] compares a traditional FS with PMFS.

3.3 Potential Modifications in a Traditional DBMS

Using a traditional disk-based database with NVM storage will not take full advantage of NVM’s features. Some important components of the DBMS need to be modified or removed when using NVM as a primary storage.

Avoid the block level access: Traditional design of a DBMS uses a disk as a primary storage. Since disks favor sequential accesses, database systems hide disk latencies by issuing fewer but larger disk accesses in the form of a data block [154].

Unfortunately, block level I/O causes extra data movement. For example, if a transaction updates a single byte of a tuple, it still needs to write the whole block of data to the disk. On the other hand, block level access provides good data locality.

Since NVM is byte-addressable, we can read and write only the required byte(s). However, reducing the data retrieval granularity down to a byte level eliminates the advantage of data locality altogether. A good compromise is to reduce the block size in such a way that the overhead of the block I/O is reduced to an acceptable level, while at the same time the application benefits from some degree of data locality.

Remove internal buffer cache of DBMS: DBMSs usually maintain an internal buffer cache. Whenever a tuple is to be accessed, first its disk address has to be calculated. If the corresponding block of data is not found in the internal buffer cache, then it is read from disk and stored in the internal buffer cache [155].

This approach is unnecessary in an NVM-based database design. If the NVM address space is made visible to a process, then there is no need to copy data blocks. It is more efficient to refer to the tuple directly by its address. However, we need an NVM-aware FS, such as PMFS, to enable direct access to the NVM address space by a process.

3.4 Discussion

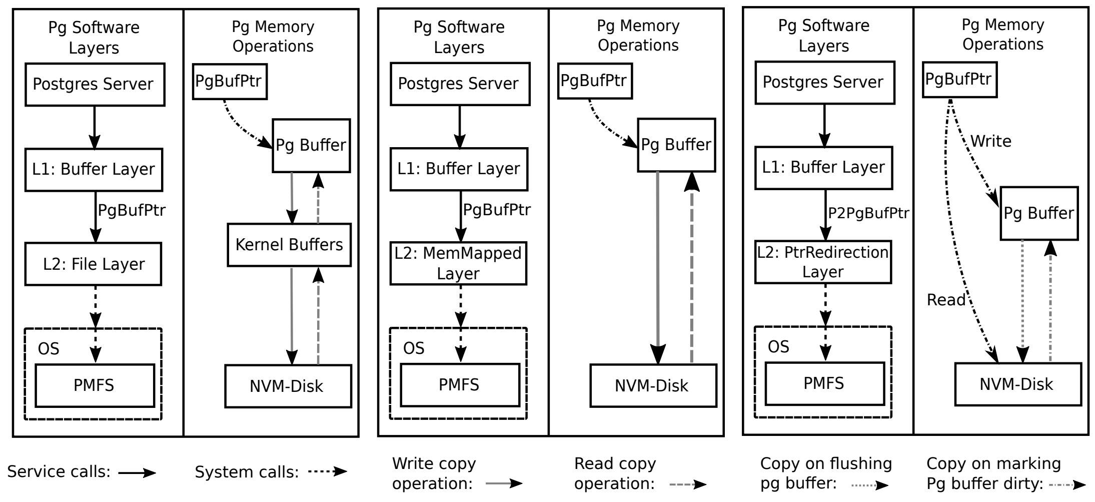
NVM provides the promising features of persistency, like disk storage; and byte-addressability, like DRAM. However, NVMs have certain limitations such as lower endurance compared to DRAM [79] and a disparity between the read and write latencies [156]. Furthermore, different NVM technologies differ from each other in term of these features [79].

An SE aiming to improve decision support system (DSS) queries can be designed by taking advantage of the common features of persistency and byte-addressability. Since DSS queries are read dominant and perform a relatively negligible number of write operations, the design should not be influenced or sensitive to different endurance and write latencies found across NVM technologies. Furthermore, NVM technologies are projected to provide read latencies similar to DRAM [47, 79, 80, 82]. Therefore, reading data directly from NVM storage should be comparable in terms of access latency to reading application data stored in DRAM.

Usage of NVM as primary storage can also impact other components of a DBMS besides those mentioned in Section 3.3. For example, if internal buffers are not employed and all updates are materialized directly into the NVM address space then the need and criticality of the redo log can be relaxed [96]. However, the undo log will still be needed to recover from a system failure. These important aspects are out of the scope of this work and we mainly focus on SE modifications.

3.5 A Case Study: PostgreSQL

PostgreSQL is an open source object-relational database system. It is fully atomicity, consistency, isolation, durability (ACID) compliant and runs on all major operating systems including Linux [157]. In this section, we explain the SE of PostgreSQL and apply necessary changes to make it NVM-aware. We first describe the read-write architecture of PostgreSQL and then explain our modifications.



(a) PostgreSQL storage engine

(b) Modified storage engine - SE1

(c) Modified storage engine - SE2

Figure 3.3: High level view of read and write memory operations in PostgreSQL (read as “pg” in short form) and modified SEs.

3.5.1 Read-Write Architecture of PostgreSQL

Figure 3.3(a) shows the original PostgreSQL architecture from the perspective of read and write file operations. The left column in the figure shows the operations performed by different software layers of PostgreSQL, while the right column shows the corresponding data movement activities. Note that in Figure 3.3(a) we assume the disk has already been replaced by NVM hardware with PMFS as the file system. However, the same behavior would be expected using a regular disk and a traditional FS, since PostgreSQL heavily relies on file I/O for read and write operations and the file I/O APIs in PMFS are the same as those in a traditional FS.

The PostgreSQL server calls the services of the *Buffer Layer* which is responsible for maintaining an internal buffer cache. The buffer cache is used to keep a copy of the requested page which is read from the storage. Copies are kept in the cache as long as they are needed. If there is no free slot available for a newly requested page then a replacement policy is used to select a victim. The victim is evicted from the buffer cache and if it is a dirty page, then it is also flushed back to the permanent storage.

Upon receiving a new request to read a page from storage, the *Buffer Layer* finds a free buffer cache slot and gets a pointer to it. The free buffer slot and corresponding pointer are shown in Figure 3.3(a) as *Pg Buffer* and *PgBufPtr*, respectively. The *Buffer Layer* then passes the pointer to the *File Layer*. Eventually, the *File Layer* of PostgreSQL invokes the file read and write system calls implemented by the underlying FS.

For a read operation, PMFS copies the data block from NVM to a kernel buffer and then the kernel copies the requested data block to an internal buffer slot pointed by *PgBufPtr*. In the same way, two copies are made for write operation but in the opposite direction.

Hence, the SE of original PostgreSQL incurs two copy operations for each miss in the internal buffer cache. This is likely to become a big overhead for databases running queries on large datasets. However, since PMFS can map the entire NVM address space into the kernel's virtual address space [152], the copy overhead can be avoided by making modifications in the SE. We apply these modifications in two incremental steps as described in the following subsections.

3.5.2 SE1: Using Memory Mapped I/O

In the first step towards leveraging the features of NVM, we replace the *File Layer* of PostgreSQL by a new layer named *MemMapped Layer*. As shown in Figure 3.3(b), this layer still receives a pointer to a free buffer slot from the *Buffer Layer*, but instead of using the file I/O interface, it uses the memory mapped I/O interface of PMFS. We term this storage engine *SE1*.

Read Operation: When accessing a file for a read operation, we first open the file using the `open()` system call, same as in original PostgreSQL. Additionally, we create a mapping of the file using `mmap()`. Since we are using PMFS, `mmap()`

returns a pointer to the mapping of the file stored in NVM. The implementation of `mmap()` by PMFS provides the application with direct access to mapped pages of files residing in NVM.

As a result, we do not need to make an intermediate copy of the requested page from NVM into kernel buffers. We can directly copy the requested page into internal buffers of PostgreSQL by using an implicit `memcpy()` as shown in Figure 3.3(b). When all requested operations on a given file are completed and it is not needed anymore, the file can be closed. Upon closing a file, we delete the mapping of the file by calling the `munmap()` function.

Write Operation: The same approach as in the read operation is used for writing data into a file. The file to be modified is first opened and a mapping is created using `mmap()`. The data to be written into the file is copied directly from internal buffers of PostgreSQL into NVM using `memcpy()`.

An SE with the above-mentioned modifications does not create an intermediate copy of the data in kernel buffers. Hence we reduce the overhead to one copy operation for each miss in the internal buffer cache of PostgreSQL.

3.5.3 SE2: Direct Access to Mapped Files

In the second step of modifications to the SE, we replace the *MemMapped Layer* of SE1 by the *PtrRedirection Layer* as shown in Figure 3.3(c). Unlike the *MemMapped Layer*, the *PtrRedirection Layer* in SE2 receives the pointer to *PgBufPtr* (i.e *P2PgBufPtr*), which itself points to a free slot of the buffer cache. In other words, *PtrRedirection Layer* receives a pointer to a pointer from the *Buffer Layer*.

Read Operation: When accessing a file for a read operation, we first open the file using `open()` system call, same as in original PostgreSQL and SE1. Additionally, we also create a mapping of the file using `mmap()`. Originally *PgBufPtr* points to a free slot in the internal buffer cache. Since `mmap()` makes the NVM mapped address space visible to the calling process, the *PtrRedirection Layer* simply redirects the *PgBufPtr* to point to the corresponding address of the file residing in NVM. Pointer redirection in case of a read operation is shown by a black dashed arrow with the “Read” label in Figure 3.3(c).

As a result of pointer redirection and the visibility of the NVM address space enabled by PMFS, we incur no copy overhead for read operations. This can represent a significant improvement since read operations are predominant in queries that operate on large datasets.

Write Operation: PMFS provides direct write access for files residing in NVM. However, it does not manage the data consistency in memory mapped operations and leaves the responsibility to the application [152] - i.e., PostgreSQL.

3.6 Data Consistency and Modified Storage Engines

PostgreSQL is an ACID compliant DBMS which uses multi-version concurrency control (MVCC) [158] to maintain data consistency. Under MVCC, concurrent executing transactions see a snapshot of the data at a particular instant in time, regardless of the current state of the underlying data. This provides data consistency and transaction isolation [159].

To keep the consistency model of PostgreSQL unaltered and functionally correct, SE2 performs two actions before modifying the actual content of the page and marking it as dirty. First, if the page is residing in NVM, it copies the page back from NVM into the corresponding slot of the internal buffer cache, i.e. *PgBuffer*. Second, it undoes the redirection of *PgBufPtr* such that it again points to the corresponding slot in the buffer cache and not to the NVM mapped file. This is shown by a black dashed arrow with the “Write” label in Figure 3.3(c). This way, SE2 ensures that each transaction (or query) updates only its local copy of the page.

In other words, SE1 and SE2 always use the internal PostgreSQL buffers for write operations, avoiding writes directly into database files residing in NVM disk. As a result, data consistency model of PostgreSQL is not violated and transactions are protected from viewing inconsistent data. In summary, SE1 and SE2 operate in the same way as far as write operation is concerned. However, for read operations, SE1 reduces overhead for each miss in internal buffer cache from two copy operations in original PostgreSQL SE to one. On the other hand, SE2 reduces this overhead to zero copy operations.

3.7 Software Library for NVM Data Prefetching

Evaluation results, explained in Section 6.1, show that modified SEs improve query execution time of PostgreSQL. However, the improvement is limited by the fact that data is not close to the processing units when needed for query processing since it is directly accessed from NVM hardware. This leads to long latency user-level cache misses that decrease the improvements achieved by avoiding expensive data movement operations.

Therefore, we employ a known technique like helper-threads for data prefetching in the context of a NVM-based database design to resolve the data readiness problem. We develop a general purpose data prefetching library that exposes a simple API allowing the creation of a user-specified number of helper threads that seamlessly prefetch data for a given address and memory size with minimum interference to the computation thread.

In following subsections, we provide details on our general purpose data prefetching library. We also explain different thread mapping schemes with and without hyper-threading.

3.7.1 Helper Threads

Prefetching data [160] reduces the cache miss rate and hence accelerates an application's execution. An in-advance knowledge of memory regions to be accessed can be used to prefetch data into caches before it is needed. However, the application should not be stalled while prefetching the data. This can be achieved by using independent helper threads for data prefetching.

Synchronization between the main computation thread and helper thread(s) is important [103]. Prefetching data too early before it is needed by the computation thread can result in cache pollution. Furthermore, required cache lines may get evicted before they are accessed. Similarly, prefetching data too late is also not useful, rather counter-productive. It can also lead to cache pollution and degrade performance.

In our implementation, a helper thread is a simple block of code. Given a starting memory address and the amount of data to be prefetched, the helper thread prefetches data into caches without interfering with the main computation thread. We employ a job queue to build a single producer - (multiple) consumer relationship between a computation thread and one or multiple helper threads. We employ light-weight compare and swap instructions for synchronization in the job queue between a computation thread and the different helper threads.

A computation thread places the starting address and the amount of data to be prefetched into a job unit and enqueues it into the job queue. On the other end, a helper thread picks the job item, unpacks it and then prefetches the data into caches. Data prefetching is performed without stalling the computation thread.

3.7.2 Library Services

A programmer is responsible for inserting API calls to construct helper threads. However, as SE2 already has knowledge of the size and location of the block to be read, inserting data prefetching APIs in SE2 source code is not a tedious task.

Our library provides three basic services via a simple API.

1. **Creation of helper threads:** The library supports the creation of a user-specified number of helper threads per computation thread, that are synchronized using a job queue. A slightly different thread creation policy is also supported, as explained in detail in Section 3.7.3.

2. **Assigning work to helper threads:** Work is assigned to helper threads by placing jobs into their job queue. On arrival of a job into the queue, helper threads wake up, one fetches the job from the queue and starts the data prefetching. After completing the job, the helper thread again waits for the next job's arrival if the queue is empty.

3. **Mapping threads to cores:** Our library also supports the selection of a core (in a multi-core platform) on which a particular helper thread is to be executed by setting thread affinities. The affinities can also be set with respect to the computational thread, i.e., selecting the same or a different core.

3.7.3 Thread Mapping Schemes

As discussed in Section 6.1, *SE2* directly accesses data located in NVM disk, without copying it into any local buffer. This direct access results in high cache miss rates when the data is needed for processing. However, due to ad-hoc data prefetching, SE2 achieves performance improvements for a few workload queries, but not for the majority of them. By performing ad-hoc data prefetching we were able to prefetch some blocks into caches, but not most of them due to the overheads it entailed performing the prefetching inside the computation thread. Furthermore, ad-hoc placement of data prefetching in the source code of an application can be tedious and difficult to maintain.

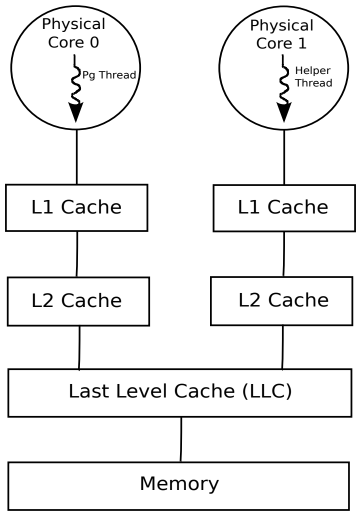
By using our data prefetching library services, data can be brought closer to processing cores before it is needed. When accessing a file for a read operation, *SE2* creates a memory mapping of the file using `mmap()`. Additionally, it has knowledge of the location and size of the data block to be read. Therefore, *SE2* can pack this information into a job and place it into the job queue to be processed by helper threads. By using this approach, PostgreSQL can continue with its computation while the required data is prefetched into caches by a helper thread.

Due to the way PostgreSQL is structured, it is not necessary to have more than one helper thread active to service a queued job in time before the next arrives. Therefore, we propose two mappings that employ only one helper thread and a third mapping that employs a different thread creation policy by instantiating two helper threads that work in tandem.

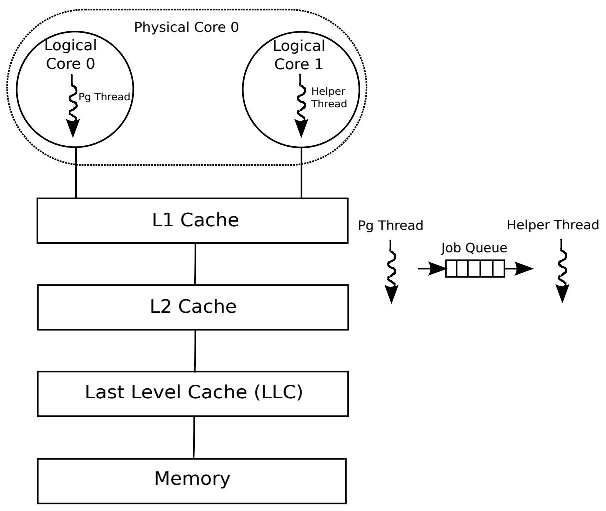
3.7.3.1 Single Helper Thread

When using a single helper thread there are two options for thread mapping. Mapping it to a different core than that of the computation thread, or to the same core. The latter option makes sense if the target machine supports hyper-threading - i.e., two hardware thread contexts per core. We explore these thread mappings:

1. `M1` - Map helper thread to a different physical core, as shown in Figure 3.4(a). In this case, each thread resides in a different core and hence prefetching is not done at the level of private caches but at the level of the last level cache, which is shared across cores.
2. `M2` - Map helper thread into the same physical core while making use of hyper-threading, as shown in Figure 3.4(b). Both threads reside within the same physical core, hence prefetching will also populate the private L1 and L2 caches present in the core.



(a) *M1* - Different physical core



(b) *M2* - Same physical core (HT)

Figure 3.4: Thread mapping schemes for a single helper thread with and without hyper-threading (HT) enabled.

3.7.3.2 Two Helper Threads

As described for case *M2*, the helper thread prefetches data into the private caches of the core where the computation thread is executing. However, in this scenario, the helper thread competes with the computation thread for hardware resources. This competition can slow down the execution of the computation thread. On the other hand, for case *M1* there is no such competition for hardware resources at the expense of prefetching into the LLC, further away from the processing core.

A good compromise can be achieved using two helper threads working together. One helper thread is mapped to a different physical core than that of the computation thread and will process the jobs enqueued by the computation thread, similar to case *M1*. Once this first thread finishes processing the job, it enqueues the same job into a second job queue that is processed by the second helper thread that is mapped into the same physical core as the computation thread. This thread mapping scheme which we term `M3` is shown in Figure 3.5.

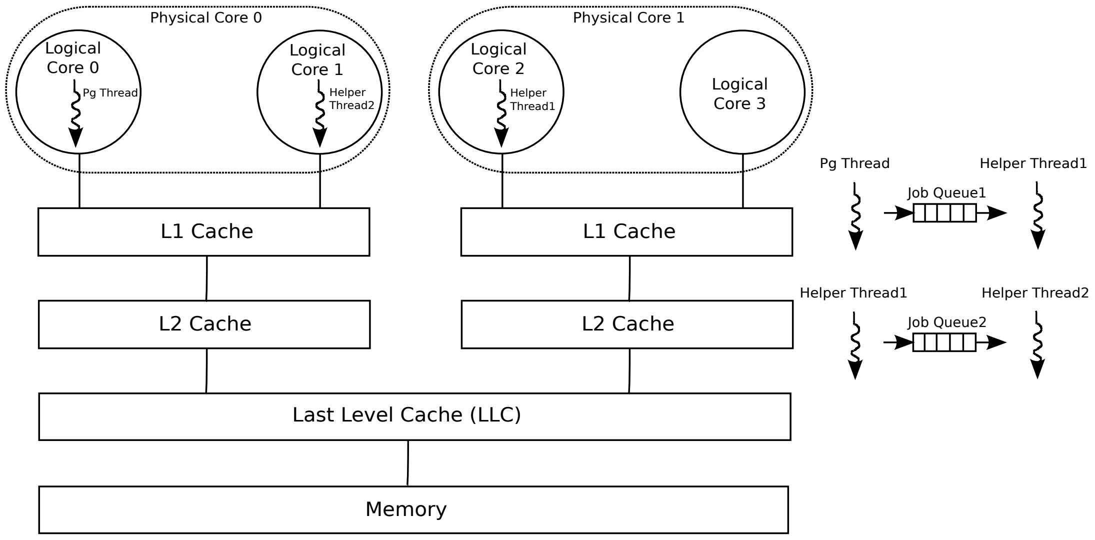


Figure 3.5: Thread mapping scheme for two helper threads.

The rationale behind this proposal is that the high penalty miss from main memory to LLC will be paid by a different core, while the helper thread residing on the same core as the computation thread will put data into the private caches and complete jobs much faster since data will be already present in the LLC.

Chapter 4

Compiler Optimization for Reduction of Memory Bound Stalls

In this chapter, we explain a compiler optimization aiming at reduction of memory bound stalls. The optimization is based on efficiently utilizing the hardware components of Myriad 2 platform. It is expected to be relatively easy to adapt the optimization for other computer vision (CV) platforms with similar architectural features. We first describe the limitations of a generic compiler in efficiently accessing the memory subsystem of Myriad 2 and then propose a solution in form of an optimization. We also explain implementation of the optimization on LLVM compiler framework.

4.1 Limitations of a Generic Compiler

In CV applications, generally, a data frame is processed by applying a filter across all of its pixels. As explained in Section 2.5.1, memory of Myriad 2 platform is organized into slices and regions. A data frame can occupy a single region of a given slice, multiple regions, or even multiple slices. Since Myriad 2 is an NUMA platform, it is important to place a data frame in a slice local to the computing processor to reduce memory latency.

Another way to reduce memory stalls is to issue multiple simultaneous memory accesses. Since SHAVE is a VLIW processor with two load store units (LSUs 0 and 1), it is a wastage of resources to perform all memory accesses in a serial fashion. Memory subsystem of Myriad 2 platform supports scheduling of up to two accesses to a given CMX slice in a single cycle provided that both LSUs are available. However, a check is required to avoid simultaneous accesses in the same region of a given slice due to the clash between memory ports, as described in Section 2.5.1. Since a base compiler (BC) does not perform this check and always schedules simultaneously requested memory accesses in the same cycle, it may result in blocking of one memory instruction by another leading to memory stalls.

Such stall cycles can be avoided by making the compiler aware of the architectural limitations of the CMX memory. If provided with the information about the physical location of each memory object in a CMX slice, the compiler can generate optimized instruction schedule leading to the reduction of MBS.

It is important to clarify that BC is assumed to be already equipped with necessary support needed to resolve conflicts among multiple SHAVE processors and/or accelerators requesting simultaneous accesses to the same memory slice. Therefore, the focus of this work is not the conflict resolution among multiple processors but handling simultaneous accesses by a single processor to the same slice.

4.2 Proposed Solution

Our proposed solution consists of two steps. In the first step, named affinity analysis, each memory object is appended with an affinity number at compile time to predict its physical location in the CMX slice.

In the second step, i.e., affinity-aware instruction scheduling (AAIS), the scheduler uses the appended affinity numbers for efficient scheduling of memory instructions. In the AAIS, two instructions requesting simultaneous access in the same CMX slice are scheduled in the same cycle only if two conditions are satisfied:

- Condition 1: Both LSUs are available
- Condition 2: Memory objects to be accessed by instructions have different affinity numbers.

If any one of these two conditions is not satisfied, instructions are not scheduled in the same cycle. In the rest of the discussion, we assume that the first condition is always true.

AAIS reduces stall cycles by avoiding blocking of one memory instruction by another. A compiler using AAIS is named as affinity-aware compiler (AAC). Otherwise, the scheduling is named as basic instruction scheduling (BIS) and the compiler as BC. Note that BC tests only the first condition for simultaneous scheduling of memory instructions, while AAC tests both of them.

To understand the calculation of affinity numbers, let us divide CMX slice into two logical vertical sections named as *Tile0* and *Tile1*. This logical division is shown in Figure 2.2(b), where each tile has two physical regions. Let us consider two instruction, $Inst1$ and $Inst2$, requesting access in the same CMX slice simultaneously with addresses for their respective memory objects as $BP+\Delta_1$ and $BP+\Delta_2$, where BP is a base address and Δ is the offset from BP. Since the base address is not known at compile time, only offset value is to be used to infer the physical location of a memory object.

Assuming that base address is 16-byte aligned (i.e., a multiple of 16), offset can be used to find the ID of the tile a memory object belongs to. Since each tile is 8-byte wide (explained in Section 2.5.1), if the offset address is in the

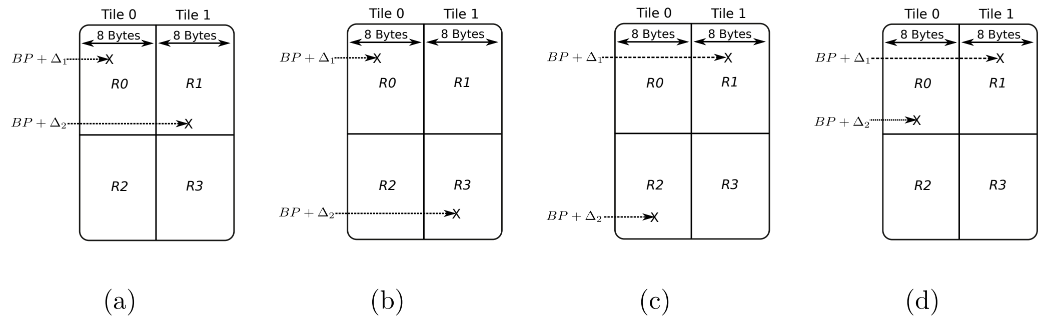


Figure 4.1: Different scenarios where two memory objects belong to different tiles. $BP + \Delta$ is the address of a memory object, where BP is the base address and Δ is the offset from BP.

range of $[8 * n, (8 * (n + 1)) - 1]$ then the memory belongs to *Tile 1*. Otherwise, the memory object belongs to *Tile 0*. With n defined as an odd number, the expression $[8 * n, (8 * (n + 1)) - 1]$ represents ranges such as $[8, ..., 15]$, $[24, ..., 31]$ and so on. Note that an offset address within these ranges will always have its third bit set to 1. In other words, using a $mask = 0x008$, a memory object with address $BP + \Delta$ belongs to *Tile 0* if bitwise *and* operation between Δ and $mask$ equals zero and to *Tile 1* otherwise.

Depending on the results of masking operations, there are two cases for the calculation of affinity numbers in the affinity analysis step.

- *Case A:* $(\Delta_1 \& mask)! = (\Delta_2 \& mask)$. In this case memory objects of $Inst1$ and $Inst2$ belong to different tiles, guaranteeing that they also belong to different regions, as shown in Figure 4.1. Since the AAIS performs a non-equality test on affinity numbers, tile IDs can be appended to memory objects in place of region IDs in the affinity analysis step without the loss of correctness.

  As the two memory objects belong to different regions and each region has its own set of memory ports, there is no architectural restriction on simultaneous execution of $Inst1$ and $Inst2$. In this case, both BIS and AAIS generate the same instruction schedule. BIS schedules $Instr1$ and $Instr2$ in the same cycle without testing the second condition. On the other

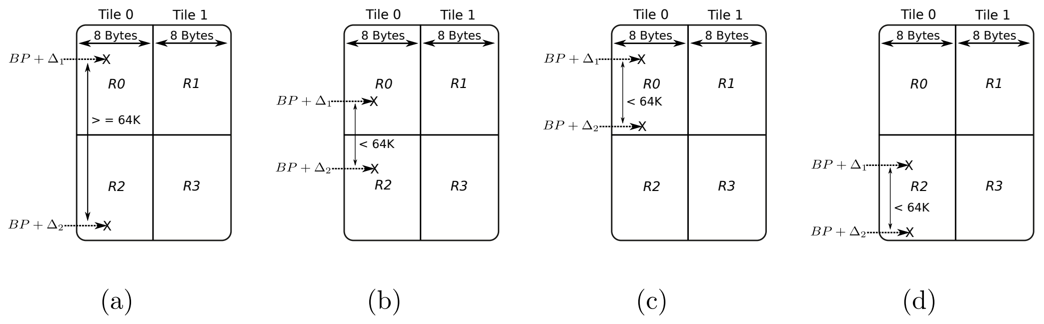


Figure 4.2: Different scenarios where two memory objects belong to the same tile. Figure 4.2(a) shows that absolute difference $Diff$ between offsets of two memory objects is greater than or equal to 64K bytes, indicating that two memory objects belong to different regions. Figure 4.2(b) shows $Diff$ less than 64K bytes and memory objects belong to different regions. Figure 4.2(c) and 4.2(d) show $Diff$ less than 64K bytes but memory objects belong to the same region.

hand, AAIS detects that second condition is true and hence schedules the instructions in the same cycle.

- *Case B:* $(\Delta_1 \& mask) = (\Delta_2 \& mask)$. In this case memory objects of $Inst1$ and $Inst2$ belong to the same tile but they may or may not belong to the same region. This can be established by calculating the absolute difference between two offsets, i.e. $Diff = abs(\Delta_1 - \Delta_2)$. An absolute difference of greater than or equal to the size of two regions (i.e., 64K bytes) guarantees that the two memory objects belong to different regions in the same tile, as shown in Figure 4.2(a). However, there is no such guarantee when $Diff$ is less than 64K bytes. Memory objects, may (e.g., Figure 4.2(b)) or may not (e.g., Figure 4.2(c) and 4.2(d)) belong to different regions in the same tile.

  Since BP is not known at compile time, it is not possible to calculate region numbers for the situations shown in Figure 4.2(b), 4.2(c), and 4.2(d). Therefore, we pessimistically assume that the two memory objects belonging to the same tile always belong to the same region. In other words, like *Case A*, we suggest appending the tile ID of a memory object as its affinity number in the affinity analysis step. In the following discussion, we break *Case B* into two sub-cases and compare the AAIS with the BIS in each sub-case.

Table 4.1: Comparison of AAIS with BIS in *Case B*.

Sch #	**BIS**	**AAIS**	***Case B1:*** Memory objects of $Inst1$ and $Inst2$ belong to the same tile but different regions.	***Case B2:*** Memory objects of $Inst1$ and $Inst2$ belong to the same tile and the same region.
1	1) IAU.ADD I10, I11, I12	1) IAU.ADD I10, I11, I12 ‖ LSU1.LD.32 I1, I8	Number of stalls does not change while number of instruction cycles reduces (from 3 to 2) in case of AAIS.	Number of stalls and instruction cycles reduce (from 1 to 0 and from 3 to 2, respectively) in case of AAIS.
	2) LSU0.LD.32 I0, I7 ‖ LSU1.LD.32 I1, I8	2) LSU0.LD.32 I0, I7 ‖ IAU.SUB I2, I3, I4	Execution time reduces (from 3 to 2 cycles) as compared to BIS.	Execution time reduces (from 4 to 2 cycles) as compared to BIS.
	3) IAU.SUB I2, I3, I4			
2	1) IAU.ADD I10, I11, I12	1) IAU.ADD I10, I11, I12 ‖ LSU1.LD.32 I1, I8	Number of stalls and instruction cycles does not change.	Number of stalls reduces (from 1 to 0) but number of instruction cycles does not change.
	2) LSU0.LD.32 I0, I7 ‖ LSU1.LD.32 I1, I8	2) LSU0.LD.32 I0, I7	Execution time remains same as in BIS (i.e 2 cycles).	Execution time reduces (from 3 to 2 cycles) as compared to BIS.
3	1) LSU0.LD.32 I0, I7 ‖ LSU1.LD.32 I1, I8	1) LSU1.LD.32 I1, I8	Number of stalls does not change but number of instruction cycles increases (from 1 to 2) in case of AAIS.	Number of stalls reduces (from 1 to 0) and number of instruction cycles increases (from 1 to 2) in case of AAIS.
		2)LSU0.LD.32 I0, I7	Execution time increases (from 1 to 2 cycles) as compared to BIS.	Execution time remains same as in BIS (i.e 2 cycles).

Case B1: Memory objects belong to the same tile but different regions. Table 4.1 compares three different BIS and AAIS-generated schedules for*Case B*. Let us assume that all three schedules are generated by BIS for the scenario depicted in Figure 4.2(a) and 4.2(b). BIS-generated instruction schedule 1 consists of one ADD instruction, two memory instructions, and a SUB instruction. With memory objects located in different regions (of the same tile), scheduling of memory instructions by BIS in the same cycle (indicated by placing $\|$ symbol between them in cycle 2) does not incur a stall. The BIS-generated schedule has an execution time of three cycles.

The corresponding AAIS-generated schedule is shown in the third column. Since memory objects belong to the same tile, the affinity analysis step pessimistically assumes that they also belong to the same region (which is not true) and appends their tile IDs as their affinity numbers. This makes the second condition to be false. As a result, AAIS serializes memory instructions by scheduling one of them in the same cycle with the ADD instruction and the other one with the SUB instruction. This reduces the execution time from 3 to 2 cycles as compared to BIS. Note that, SHAVE is a VLIW processor, capable of executing multiple instructions in the same cycle depending on the availability of functional units.

The BIS-generated instruction schedule 2 consists of one ADD instruction and two memory instructions. The schedule incurs no stall cycle and has an execution time of 2 cycles. As shown in the third column, AAIS serializes the two memory instructions by scheduling one of them with the ADD instruction. Although it changes the schedule as compared to BIS, the execution time remains same as there is no change in the number of instruction cycles and stalls.

The BIS-generated instruction schedule 3 consists of only two memory instructions. The schedule incurs no stall cycle and has an execution time of 1 cycle. On the other hand, AAIS serializes the two memory instructions by scheduling them in different cycles. This increases the execution time from 1 to 2 cycles as compared to BIS.

In summary, AAIS in *Case B1* does not change the stall cycles as compared to BIS. However, it may increase, decrease or not affect the number of instruction cycles depending on the BIS-generated instruction schedule.

Case B2: Memory objects belong to the same tile and the same region. Let us assume that all BIS schedules shown in Table 4.1 are generated for the case depicted in Figure 4.2(c) and 4.2(d). With memory objects located in the same region (of the same tile), the two memory instructions scheduled in the same cycle by BIS in schedule 1 incur a stall. This results in total execution time of 4 cycles (i.e. three instruction cycles and one stall). In the corresponding AAIS-generated schedule, affinity analysis step assumes that memory objects belonging to the same tile also belong to the same region (which is true in this case). As a result, AAIS serializes memory instructions by scheduling them in different cycles (with ADD and SUB instructions). This not only avoids the stall cycle but also reduces the instruction cycles. Hence, the execution time reduces from four to two cycles as compared to BIS.

The BIS-generated instruction schedule 2 incurs one stall cycle for two memory instructions scheduled in the same cycle. This results in an execution time of three cycles, i.e., two instruction cycles and one stall. However, AAIS serializes the two memory instructions by scheduling one of them with the ADD instruction. This saves the stall cycle and hence reduces the execution time from 3 to 2 cycles.

The BIS-generated instruction schedule 3 consists of only two memory instructions and incurs a single stall cycle, resulting in an execution time of two cycles. On the other hand, AAIS serializes the two memory instructions by scheduling them in different cycles. This avoids the stall cycle but also increase the number of instruction cycles resulting in no effect on the execution time.

In summary, AAIS in *Case B2* reduces the stall cycles as compared to BIS. However, it may increase, decrease or not affect the number of instruction cycles depending on the BIS-generated instruction schedule.

It is clear from the above discussion that AAIS can reduce stall cycles when simultaneous accesses are requested into the same region of a tile. Furthermore, above discussion shows that tile IDs can be used as affinity numbers without the loss of correctness. It is important to mention that affinity numbers are merely used as a compile time prediction for the physical location of memory objects and do not provide any means of controlling data placement.

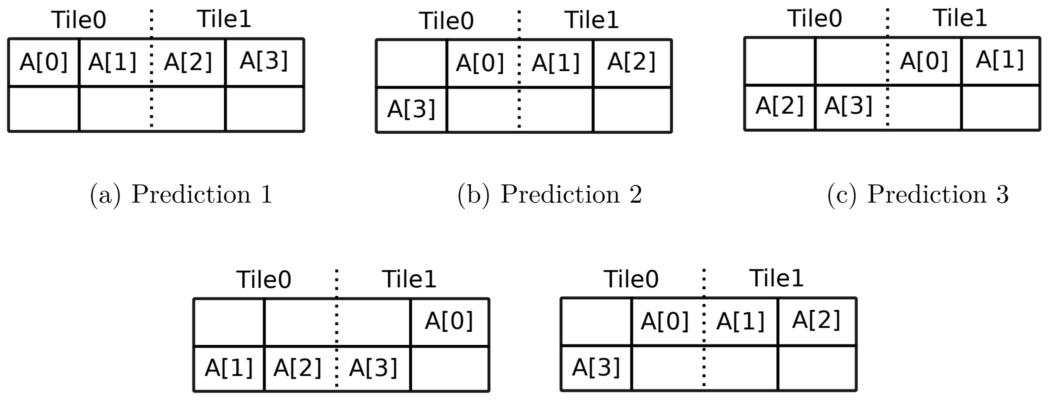
We propose two different approaches for the first step of our solution, i.e., affinity analysis. In the source code annotation approach, tile IDs are appended to memory objects by a programmer using custom attributes. In the automated analysis approach, tile IDs are inferred from the source code by analyzing the relative addresses of memory objects. Note that any one of these two approaches can be combined with the second step (i.e., AAIS) to construct the complete solution. Both approaches are discussed below.

4.2.1 Source Code Annotation

It is a compile time approach that involves appending the tile ID to each memory object in the application source code. This is achieved by defining a custom attribute and making the compiler aware of its syntax and semantics.

In annotation based approach, the difference between predicted and actual physical locations of memory objects depends upon the knowledge a programmer has about the layout of application data. Tile IDs can be appended more accurately by having a good understanding of data structures used in the application and their access patterns. For example, tile IDs to be appended to array elements depend upon the location of the first element in a CMX slice (i.e. Tile 0/1), offset from the base address and the byte-alignment.

Figure 4.3 shows different predictions for physical locations of array elements by appending tile IDs through source code annotation. The actual placement of array elements in the CMX slice is shown in Figure 4.3(e). Let us assume that the BC simultaneously schedules the access to A[0] in the same cycle with A[3] and access to A[1] in the same cycle with A[2]. This will result in two stall



(a) Prediction 1 (b) Prediction 2 (c) Prediction 3

(d) Prediction 4 (e) Actual placement

Figure 4.3: Predictions of physical locations through source code annotation.

cycles as two memory accesses will be blocked due to a clash between memory ports. In other words, this particular example has *optimization potential* of 2. Figure 4.3(b) shows the best prediction of physical locations as it reflects the real mapping of array elements in the CMX slice.

4.2.2 Automated Analysis

To avoid the modification of application source code and to automate the process of appending tile IDs to memory objects, we also propose the automated analysis. It is also a compile time approach like source code annotation. Since the base address of a memory object is not known at compile time, the automated approach uses relative addresses for calculating tile IDs with the following assumptions about data storage.

1. Data structures are stored in memory in a sequential manner.
2. The first element of a data structure is always located at the 16-byte boundary.

We propose Algorithm 1 for automated analysis of addresses and appending

Algorithm 1: Dynamic Tile-Affinity Allocation

```
Input: Function Name Func.
Output: Tile affinity allocated to all memory references belonging to the
        same data structure in Func.
1  for each instruction i in Func do
2      if (i is a memory access instruction) then
3          memObject_i = getAdrOprnd(i);
4          [BA_memObject_i, Offset_memObject_i] =
            getInfo(memObject_i);
5          if (hasNoAffinityAllocated(memObject_i)) then
6              memObject_i.setAffinity(
                (0x08)&(Offset_memObject_i)?1:0);
7          end
8      end
9      for each instruction j in Func do
10         if (j is a memory access instruction) then
11             memObject_j = getAdrOprnd(j);
12             [BA_memObject_j, Offset_memObject_j] =
                getInfo(memObject_j);
13             if (BA_memObject_i == BA_memObject_j) && (i ≠ j)
                && (hasNoAffinityAllocated(memObject_j)) then
14                 memObject_j.setAffinity(
15                 (0x08)&(Offset_memObject_j)?1:0);
16             end
17         end
18     end
19 end
```

tile IDs to memory objects. It is applicable only to data elements belonging to the same data structure and it operates on the level of each function independently.

Algorithm 1 operates as follows. For each memory instruction i in the function *Func*, the address of its memory object is retrieved into *memObject_i* (lines 1 to 3) and then broken into the base address and the offset (line 4). If not already appended with a tile ID, one is calculated for *memObject_i* (lines 5 to 7). Note that the width of each tile of a CMX slice is 8 bytes, as shown in Figure 2.2(b). Tile ID is decided based on the masking of offset value (i.e. $Offset\_memObject\_i$) with 0x008. The inner for loop (line 9 to 17) scans all other memory instructions in *Func* to find if they access a memory object with the same base address as *memObject_i* but with a different offset value. The condition of same base address ensures that memory objects belong to the same data structure. If the condition is true and new memory object (i.e. *memObject_j*) is not already appended with a tile ID, it is appended with a one based on masking of its offset value (i.e. $Offset\_memObject\_j$) with 0x008.

4.2.3 Discussion

AAIS leads to the reduction of memory stalls only if an application has an inherent potential for optimization. If there are no simultaneous requests for accessing the same region of a CMX slice, there will be no stall cycles in the BIS-generated schedule and hence no space for AAIS to optimize the schedule. The *optimization potential* of an application can be defined as the number of simultaneous memory accesses to the same region scheduled in a single cycle by BIS. *Optimization potential* can easily be calculated by inspecting the assembly code of an application generated by the BC.

To harness the *optimization potential* of an application, physical locations of memory objects predicted by affinity numbers and their actual physical locations should be same. Otherwise, appended tile IDs will provide wrong information to the scheduler for AAIS leading to the generation of an unoptimized or unwanted instruction schedule.

The source code annotation places a burden on the programmer to attach custom attributes to memory objects. For large applications, it can be time-consuming and may also require modifications other than simply attaching custom attributes. However, if data placement can be enforced on the application data and it is a known priori, this approach could be more accurate and beneficial in reducing stall cycles than the automated one.

On the other hand, automated analysis relieves the programmer from the manual modifications in the application source code. However, it appends tile IDs to memory objects based on their offset from the base address of data structure they belong to. It also assumes that the first element of a data structure is always aligned at the 16-byte boundary. This may result in appended tile IDs (i.e. predicted physical locations) not reflecting the actual locations of memory objects.

4.3 Implementation

In this section, we provide the details of modifications applied on Myriad 2 compiler in order to implement the proposed solution. We first explain the implementation of affinity analysis step using both approaches (i.e., source code annotation and automated affinity analysis) followed by the implementation details of AAIS.

4.3.1 Affinity Analysis Through Source Code Annotation

Myriad 2 compiler is an extended version of LLVM compiler framework that is tailored to generate code for a SHAVE processor. Like LLVM, Myriad 2 compiler also uses Clang as a compiler front-end for C/C++ languages. To implement the annotation based approach, we modify the LLVM framework at two levels.

4.3.1.1 Adapting the Front-End

The idea of source code annotation is to allow the programmer appending tile IDs to all memory objects in the application source code. We enable this by defining a custom attribute for any type and making Clang aware of its syntax and semantics. We modify the Clang source code as described below.

1. Add the definition of the custom attribute to Clang.
2. Modify the relevant functions in Clang to detect if a given variable declaration or initialization in the source code has the custom attribute defined

in step 1. The variable can be of any type including the pointer variable.

3. If a variable has the custom attribute, then attach the metadata to the corresponding *alloc*, *load*, or *store* instructions generated for allocation or initialization of the variable.

4.3.1.2 Adapting the Back-End

Above mentioned modifications enable the Clang to recognize custom attribute and take the appropriate actions to process it. However, the information needs to be propagated from the front-end to the back-end.

In LLVM, selectionDAG builder class builds an initial directed acyclic graph providing an abstraction for code representation [161]. We modify the relevant functions in the class to propagate the predicted physical locations of memory objects down to the post-register allocation scheduling (PostRAS) pass. This is achieved by detecting the existence of metadata of the desired kind while visiting *alloc*, *store*, and *load* instructions. Upon finding metadata, tile IDs are appended to memory objects of these instructions based on the value of their metadata.

4.3.2 Affinity Analysis Through Automated Analysis

Unlike the source code annotation, the automated analysis does not necessitate the modifications in front-end and the Selection DAG builder class of the back-end. We implement the automated analysis by writing a custom LLVM pass, named as “Address Analysis Pass” (AAP), shown in Figure 4.4. AAP is invoked after register allocation and the generation of a basic instruction schedule. It calculates and appends tile IDs to memory objects by implementing Algorithm 1 and needs to be executed before the SHAVE PostRAS pass. Once the tile IDs are appended, SHAVE PostRAS pass creates affinity-aware instruction schedule, as described in the following subsection.

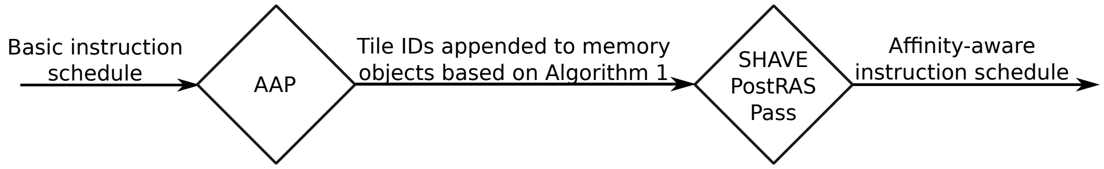

Basic instruction schedule
AAP
Tile IDs appended to memory objects based on Algorithm 1
SHAVE PostRAS Pass
Affinity-aware instruction schedule

Figure 4.4: Implementation of automated analysis as a custom pass in LLVM. Address analysis pass (AAP) appends affinity numbers to memory objects using Algorithm 1. Modified SHAVE PostRAS pass creates affinity-aware instruction schedule based on affinity numbers.

4.3.3 Affinity-Aware Instruction Scheduling (AAIS)

In LLVM framework, the compiler consists of multiple passes which perform particular transformations and optimizations. In Myriad 2's BC, a basic instruction schedule is generated by preceding passes before AAP as shown in Figure 4.4. We modify PostRAS pass to update basic instruction schedule based on tile IDs appended to memory objects in AAP. The modified PostRAS pass detects the conflicts among memory instructions by comparing tile IDs of their memory objects (i.e., testing the second condition defined in Section 4.2) and saves stall cycles by not scheduling them in the same clock cycle.

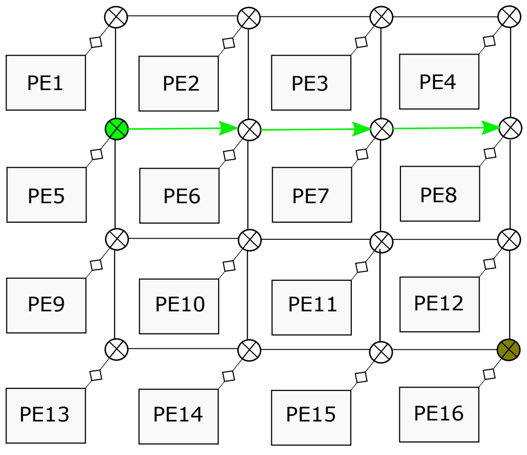
Chapter 5

A Selection Scheme for NoC Routing Algorithms

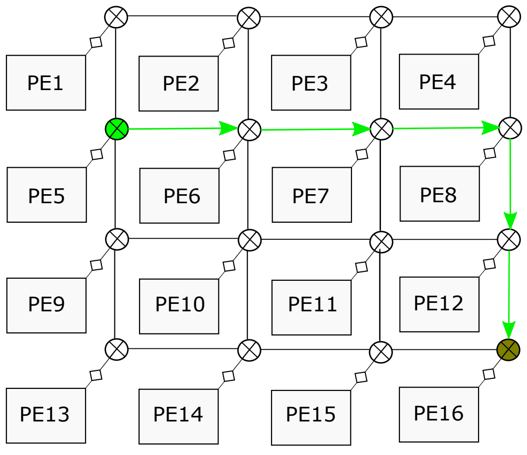
In this chapter, we first explain different routing algorithms for a mesh-based NoC. Then, we describe traffic patterns and their affect on throughput delivered by a routing algorithm. Finally, we address the issue of improving NoC throughput in the presence of changing traffic pattern.

5.1 Routing Algorithms

A routing algorithm can be categorized as deterministic, partially adaptive or fully adaptive one. A deterministic algorithm routes data packets along a fixed path between a source and a destination pair [162]. For example, XY routing [41] first routes data packets from the source PE to destination PE only along the X direction. Once the X direction of data packet is aligned with that of destination PE, the algorithm routes the packet along Y direction. Figure 5.1 demonstrates XY routing of data packets from PE5 to PE16. As XY routing algorithm provides only a single path between a pair of source and destination PE, it performs poor at balancing the data traffic across an NoC in case of network congestion [45].



(a)



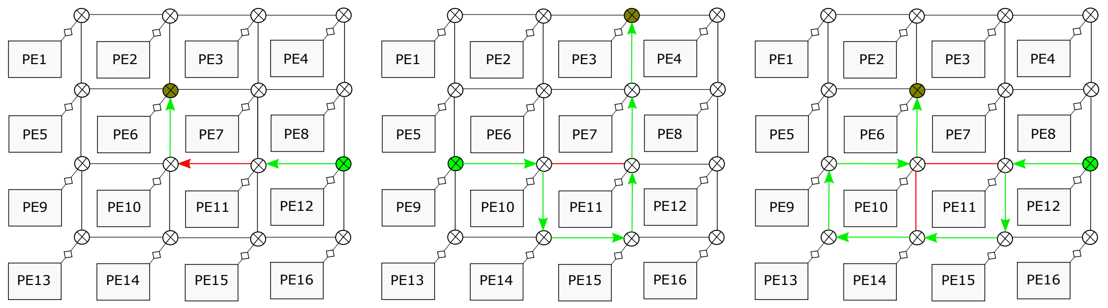
(b)

Figure 5.1: XY Routing algorithm first routes data packets from source to destination along (a) X direction and then along (b) Y direction.

Partially adaptive algorithms improve traffic balancing in case of congestion by providing multiple paths between a pair of a source and destination PE [45]. However, in order to avoid deadlock, they restrict turns a packet may take on its path toward the destination. Deadlock is a situation when data packets can not make progress towards their destination as they wait on each other to release the resources needed for progress.

Examples of partially adaptive algorithms are west-first, north-last and negative-first [42] routing. In west-first routing, a data packet must move first in the west direction. Once it takes a turn from west direction, it may move in any other direction except west. In north-last routing, a data packet may move freely among directions except north. Once it takes a turn toward north direction, it must continue in the same direction until it reaches the destination. In negative-first algorithm, a data packet must take all of its turns first in negative directions (i.e. west and south) before turning to a positive direction. Once in a positive direction, a packet must move towards its destination without taking a negative turn. Figure 5.2 demonstrates west-first, north-last and negative-first routing.

A fully adaptive routing algorithm, such as DyXY [43], makes the routing



(a) (b) (c)

Figure 5.2: Routing of data packets through (a) west-first (b) north-last and (c) negative-first algorithms. Red colored links are congested ones while green color indicates taken path.

decision based on network conditions (e.g. congestion) [162] and routes packet through any path to the destination without restricting taken paths [163]. Some algorithms, such as DyAD [44], combine deterministic and adaptive routing.

5.2 Traffic Patterns

A traffic pattern represents the distribution of data packets in an NoC and has two aspects: Spatial and Temporal. Spatial distribution indicates a set of destination PEs for a given source PE. On the other hand, temporal aspect indicates average number of packets per cycle (also known as injection rate) a source PE sends to a given destination.

A traffic pattern may be categorized as synthetic or realistic one. Synthetic traffic patterns are abstract models of message passing in NoCs whereas realistic traffic patterns are traces of real applications running on NoCs [164]. A uniform random traffic pattern distributes data traffic uniformly across an NoC as each source is equally likely to send packets to each destination [165]. A non uniform pattern concentrates traffic on individual source-destination pairs and is more closer to traffic pattern of real applications [166]. As an example of non-uniform pattern, in a 15x15 mesh, transpose1 traffic is constructed by sending packets from a source PE at index (i, j) (i.e., row i and column j) only to a single destination at index $(14 - j, 14 - i)$ [163].

5.2.1 Effect of Traffic Patterns on Performance of Routing Algorithms

While a traffic pattern determines the communication pattern, a routing algorithm is responsible for determining the communication path. Therefore, performance of a routing algorithm, among other factors, is also affected by traffic patterns. For example, as shown in [163], XY routing algorithm outperforms partially adaptive algorithms (i.e. negative-first and west-first) under uniform traffic pattern. However, negative-first algorithm performs best among XY, west-first and negative-first routing under transpose1 traffic, i.e., a synthetic non-uniform traffic pattern.

In a real application scenario, tasks mapped on different PEs (connected through an NoC) may communicate at different data rates with data injected at different time points. As a result, traffic pattern may change during execution period of an application. For example, a data-producer task may wait for an external event or a computation to be completed before it can send data to a consumer task. Therefore, it is promising to switch routing algorithms with the changing traffic pattern. In the following sections, we present a selection scheme aiming at selecting an appropriate algorithm. We propose two methods for selection, namely, static selection and dynamic selection.

5.3 Static Selection

We perform static selection in two phases. In the first phase, we apply all candidate algorithms on an application individually and obtain values for its performance metric. In this work, performance metric is throughput and candidate algorithms are XY, west-first and negative-first.

In the second phase, we find “Switching Points” and “Next Algorithm” for each switching point by analyzing the performance graphs generated using measurements obtained in first phase. “Switching Points” are the time instants at which a new algorithm (other than one currently active) is to be selected. We select switching points and next algorithm in such a way that new performance is higher than that given by candidate algorithms individually.

In static selection, switching points and their corresponding next algorithms are determined offline. Additionally, selection is made at NoC level, which means that all routers execute same routing algorithm. We elaborate on static selection in CHapter 6 with simulation results for real applications.

5.4 Dynamic Selection

Performance of an algorithm depends upon congestion level of the network at a given time and adaptivity of that routing algorithm for a particular application. Therefore it is more logical to make dynamic selection of routing algorithms based on their adaptivity and congestion level of the network. Adaptivity of an algorithm is the number of shortest paths it allows for a given source and destination pair [42].

Dynamic selection can be made either at the level of regions or routers. In the case of regional selection, congestion can be measured by using one of the three techniques described in [167]. However, for the sake of simplicity, in this work we select algorithms at the router level. As shown in Figure 5.3, we measure congestion level (CL) at a router as the total length of occupied FIFOs for that router.

For a given application, we first define an adaptivity list (AL) of algorithms for that application and a set of congestion threshold points (CTPs). AL is a listing of candidate algorithms in such a way that the adaptivity of algorithm i for the given application is less than that for the algorithm $i + 1$. N number of

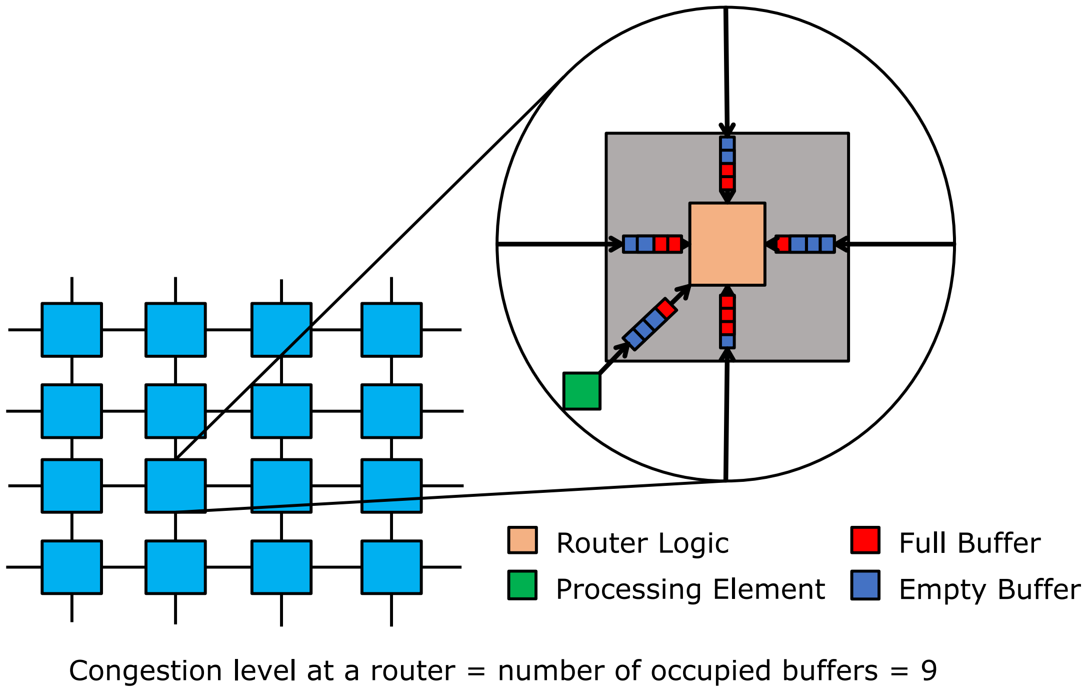

Router Logic
Processing Element
Full Buffer
Empty Buffer
Congestion level at a router = number of occupied buffers = 9

Figure 5.3: An NxN mesh: congestion level at a router is a measure of occupancy level of input FIFOs.

CTPs are defined using trial and error method, where N is the total number of candidate algorithms. CTPs are ordered in such a way that $CTP_i < CTP_{i+1}$. Algorithm 2 describes the selection of routing algorithms based on CL, CTPs and AL. Algorithm 2 operates as follows.

The algorithm maintains a variable i to keep record of congestion level (line1). At each cycle (and at each router), if i is less than total number of congestion threshold points CTP (line 2 and 3) then congestion level CL is calculated. If CL is less than current threshold point $CTP[i]$ (line 4) than same algorithm continues routing data (line 5). However, if CL is greater than current threshold point $CTP[i]$ but less than or equal to next threshold point $CTP[i+1]$ then next candidate algorithm $AL[i+1]$ is selected (line 6 and 7). At the same time i is updated (line 8). Application of Algorithm 2 is demonstrated while evaluating the proposed selection scheme in Section 6.3.

In dynamic selection, different routers may run different routing algorithms simultaneously. A router can switch to next routing algorithm based on its local

Algorithm 2: Switches routing algorithm based on congestion level

```
Input: $CL, CTP, AL, N$
1  $i \leftarrow 0$
2  for every simulation cycle do
3      if $i < N$ then
4          if $CL \leq CTP[i]$ then
5              Switch routing algorithm to AL[i].
6          else if $CTP[i] < CL \leq CTP[i+1]$ then
7              Switch routing algorithm to AL[i+1].
8              $i \leftarrow i + 1$
9          end
10     end
11 end
```

congestion level information.

Chapter 6

Evaluation

In this chapter we present methodologies and evaluation results for all of our three proposals described in Chapter 3, 4 and 5. Specifically, Section 6.1 focuses on evaluation of modified PostgreSQL storage engines (SE), whereas Section 6.2 presents evaluation of proposed compiler optimization. Finally, Section 6.3 describes evaluation results of proposed selection scheme for NoC routing.

6.1 Evaluation of NVM-Aware Storage Engines: SE1 and SE2

In this section we first describe an emulation platform and the workload used to evaluate our proposal for integration of NVM in memory hierarchy of a DBMS as a storage medium. We then present evaluation results for modified SEs of PostgreSQL. Since we use data prefetching to solve readiness problem, we also present performance evaluation of modified SE when using data services of data prefetching library described in Section 3.7.

6.1.1 Methodology

System-level evaluation for NVM technologies is challenging due to lack of real hardware. Software simulation infrastructures are a good fit to evaluate systems in which NVM is used as a DRAM replacement, or in conjunction with DRAM as a hybrid memory system. However, when using NVM as a permanent storage replacement, most software simulators fail to capture the details of the operating system, and comparisons against traditional disks are not possible due to the lack of proper simulation models for such devices. As the authors of PMFS [152] noted, an emulation platform is the best way to evaluate such a scenario.

6.1.1.1 Emulation Platform

We set up an infrastructure similar to that used by the PMFS authors. We first recompile the Linux kernel of our test machine with PMFS support. Using the *memmap* kernel command line option we reserve a physically contiguous area of the available DRAM at boot-time, which is later used to mount the PMFS partition. In other words, a portion of the DRAM holds the disk partition managed by PMFS and provides features similar to those of NVM, such as byte-addressability and lower latency compared to a disk. Table 6.1 lists the test machine characteristics. We configure the machine to have a 224GB PMFS partition, leaving 32GB of DRAM for normal main memory operation. A high-end SSD is used as regular disk storage.

To fairly evaluate SE1 and SE2, we compare their performance with two baselines using unmodified PostgreSQL (following the same comparison approach as adopted by other closely related and complementary works [99, 100]). The two baselines use unmodified PostgreSQL 9.5 with the dataset stored in: (i) a regular high-end disk (*disk_base95*), and (ii) in the PMFS partition (*pmfs_base95*). The modified storage engines - SE1 and SE2 - are run with the dataset stored on the PMFS partition and are termed *pmfs_se1* and *pmfs_se2*, respectively.

Since DRAM read latencies are expected to be similar to projected NVM

Table 6.1: Test machine characteristics.

Component	**Description**
Processor	Intel Xeon E5-2670 @ 2.60Ghz HT and TurboBoost disabled
Caches	Private: L1 32KB 4-way split I/D, L2 256KB 8-way Shared: L3 20MB 16-way
Memory	256GB DDR3-1600, 4 channels, delivering up to 51.5GB/s
OS	Linux Kernel 3.11.0 with PMFS support [152, 153]
Disk storage	Intel DC S3700 Series, 400GB, SATA 6Gb/s Read 500MBs/75k iops, Write 460MBs/36k iops
PMFS storage	224 GB of total DRAM

read latencies [47, 79, 80, 82], the emulation platform we employed provides good performance estimations. In our experiments, we report wall-clock query execution times as well as data obtained with performance counters using the *perf* toolset.

6.1.2 Workloads

TPC-H [53] is a widely used benchmark and a good representative of decision support system (DSS) queries. Therefore, to evaluate our proposed SEs, we employ DSS queries from the TPC-H benchmark configured with a scale factor of 100, which leads to a dataset larger than 150GB when adding the appropriate indexes. Like most data-intensive workloads, these queries are read dominant, which will enable us to draw accurate results from our emulation platform. We report results for 16 of the 22 TPC-H queries since some queries fail to complete under PMFS storage, even when executed with the unmodified PostgreSQL SE (i.e. baseline *pmfs_base95*).

Figure 6.1 shows the characterization of the different TPC-H queries in the

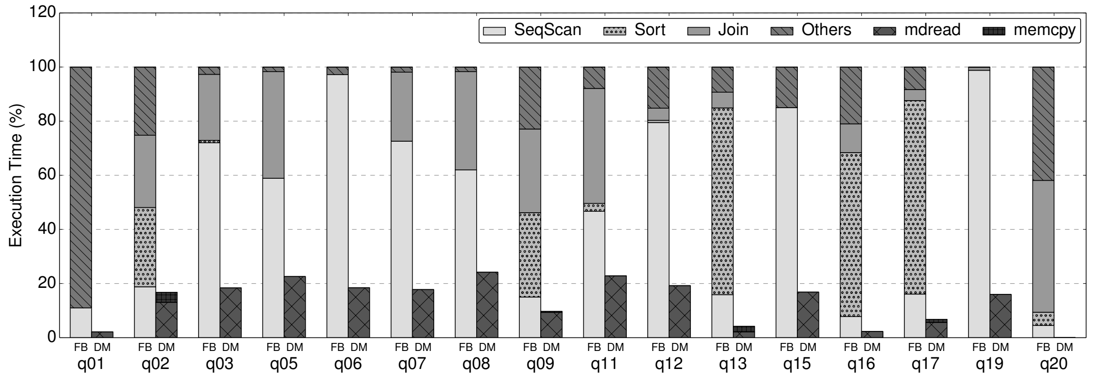


Figure 6.1: Execution time breakdown for TPCH queries in traditional DBMS with database stored in disk-storage.

form of an execution time breakdown. The data is collected using a scale factor of 100 with a baseline system that uses a high-end disk as primary storage (*disk_base95*). The figure shows two bars: functional breakdown (FB) and data movement (DM). FB shows the percentage of execution time spent across the most relevant database operators, i.e., sequential scan (SeqScan), Sort, Join and all other operations combined together. DM shows the percentage of execution time spent in the main function performing data reads from disk (*mdread*) and also *memcpy* since it is used internally by the kernel to bring data into the application buffers.

As can be seen in the figure, most of the queries are dominated by sequential scan operations, as expected from read-dominant queries. This is confirmed by the fact that most queries spend about 20% of their execution time bringing data in from storage to application-level buffers, as shown by the DM bar. These overheads are expected to become worse with larger datasets in the future, therefore lowering the data access latency and avoiding unnecessary data movement is critical to reduce query execution time. Note that these overheads are due to data movement operations that can be avoided by reading directly from primary storage with our proposed NVM-aware SEs.

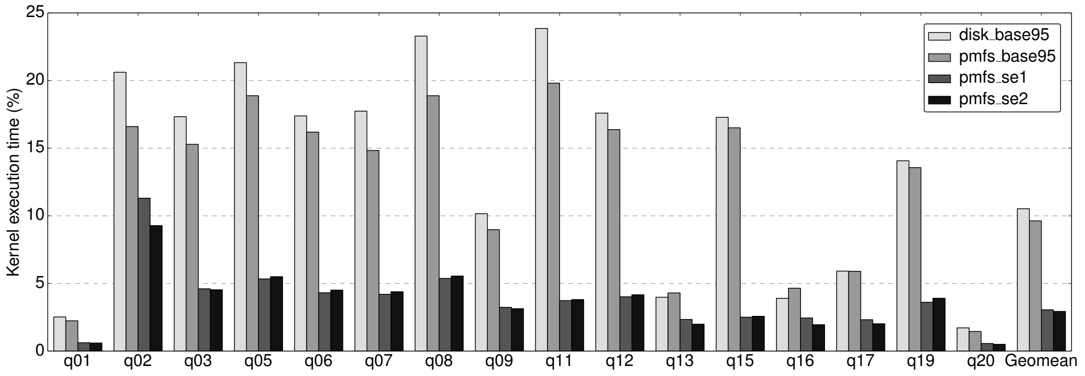


Figure 6.2: Percentage of kernel execution time for each query.

6.1.3 Performance Evaluation of SE1 and SE2

In this subsection, we show the performance impact of the modified SE on kernel execution time and on wall-clock execution time for TPC-H queries. Later, we identify potential issues current DBMSs and applications, in general, may face in order to harness the benefits of directly accessing data stored in NVM memory.

6.1.3.1 Performance Impact on Kernel Execution Time

Figure 6.2 shows the percentage of kernel execution time (KET) for each of the evaluated queries running on the four evaluated systems. When using traditional file operations (e.g. `read()`), like those employed in unmodified PostgreSQL, the bulk of the work when accessing and reading data is done inside the kernel. As can be seen, the baseline systems spend a significant amount of the execution time in kernel space: up to 23.85% (Q11 - *disk_base95*) and 19.80% (Q11 - *pmfs_base95*), with an average of around 10%. KET is dominated by the time it takes to fetch data from the storage medium into a user-level buffer. These overheads are high in both disk and NVM storage, and are likely to increase as datasets grow in size.

However, when using *SE1* or *SE2*, this data movement can be minimized or even avoided. For *pmfs_se1* we observe that the amount of time spent in kernel

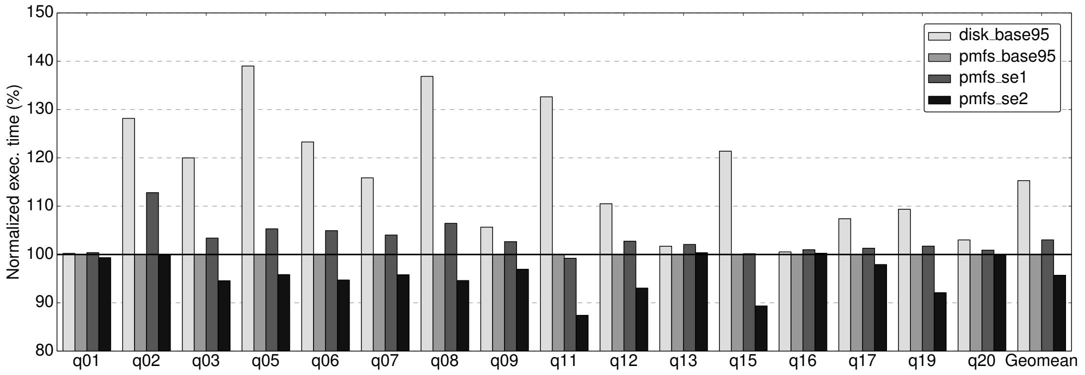


Figure 6.3: Wall-clock execution time normalized with respect to *pmfs_base95*.

space decreases substantially and it is very similar to that observed for *pmfs_se2*. This is because the two systems are doing a similar amount of work on the kernel side, with the difference that *SE1* is doing an implicit `memcpy()` operation into a user-level buffer. Overall, we see that the modified SEs are able to reduce KET significantly in most queries: Q02 to Q12, Q15, and Q19. A few queries show lower reductions because they operate over a small amount of data, e.g, Q01, Q13, Q16, and Q20.

Reduction in KET is consistent with the data movement (DM) bar in Figure 6.1. For example, queries with relatively more time spent in DM operations (e.g., Q02 to Q12, Q15, and Q19) show a higher reduction in KET when executed using SE1 and SE2, as shown in Figure 6.2. On the other hand, queries that spend relatively lesser time in DM operations (e.g., Q01, Q13, Q16, and Q20) show a lower reduction in KET. In other words, read intensive queries show more reduction in KET when executed using SE1 and SE2. An important point to note is that for *SE1* and *SE2* the kernel space time is likely to remain near constant as datasets grow since no work is done to fetch data.

6.1.3.2 Query Performance Improvement

Figure 6.3 shows wall-clock execution time for each query on the evaluated systems. The data is normalized with respect to *pmfs_base95*. We observe that the benefits of moving from disk to a faster storage can be high for read-intensive queries such as Q05 (39%), Q08 (37%), and Q11 (33%). However, for compute-intensive queries, such as Q01 and Q16, the benefits are non-existent. On average, the overhead of using disk over PMFS storage is around 15%.

For *SE1*, the reductions observed in terms of kernel execution time do not translate into reductions in overall query execution time. The main reason for this is the additional `memcpy()` operation performed to copy the data into the application buffer. In fact, we find that this operation in PMFS is sometimes slower than the original `read()` system call employed in the baseline, leading to a 3% slowdown on average.

When using *SE2* there is no data movement at the time of fetching data into an application-accessible memory region, due to the possibility of directly referencing data stored in PMFS. However, this has a negative side effect when accessing the data for processing later on, as it has not been cached by the processing units. Therefore, the benefits of avoiding data movement to make it accessible are offset by the penalty to fetch this data close to the processing units at a later stage. In order to mitigate this penalty, *SE2* incorporates a simple software prefetching scheme that tries to fetch in advance the next element to be processed within a data block.

When compared to *pmfs_base95*, *SE2* is able to achieve significant performance improvements in read-dominant queries such as Q11 (13%), Q15 (11%), and Q19 (8%). These performance improvements are also consistent with the data presented in Figure 6.1. Queries that are dominated by sequential scan operations are the ones that benefit from our modified SE. For example, although Q02 spends almost 18% of its execution time in data movement operations (like Q15 and Q19, as shown by the DM bar in Figure 6.1), sequential scan makes only 20% of the database operations performed by the query (unlike Q15 (84%) and Q19 (98%)) as shown by the FB bar in the same figure. As a result, Q02 does not benefit from our modified SE. The same explanation is valid for Q13, Q16, Q17, and Q20. On average, *SE2* is around 4% faster than *pmfs_base95* and around 19% faster than *disk_base95*.

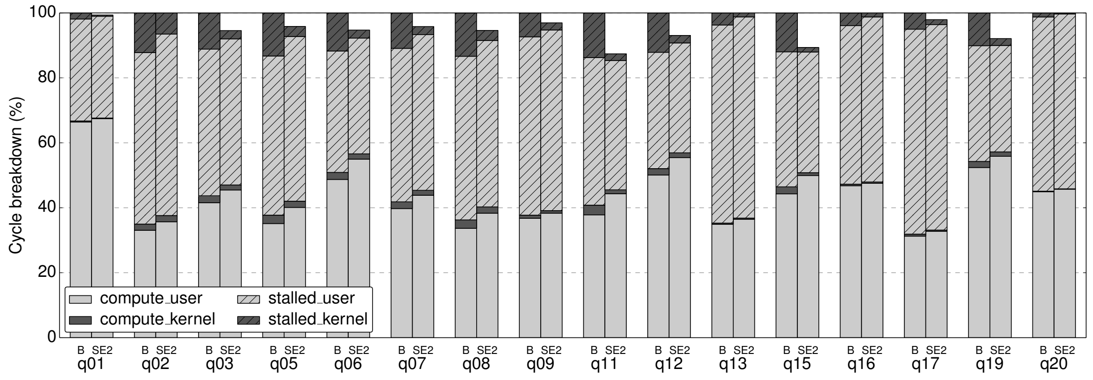


Figure 6.4: Execution-time breakdown for compute and stalled cycles — B = *pmfs_base95*, SE2 = *pmfs_se2*.

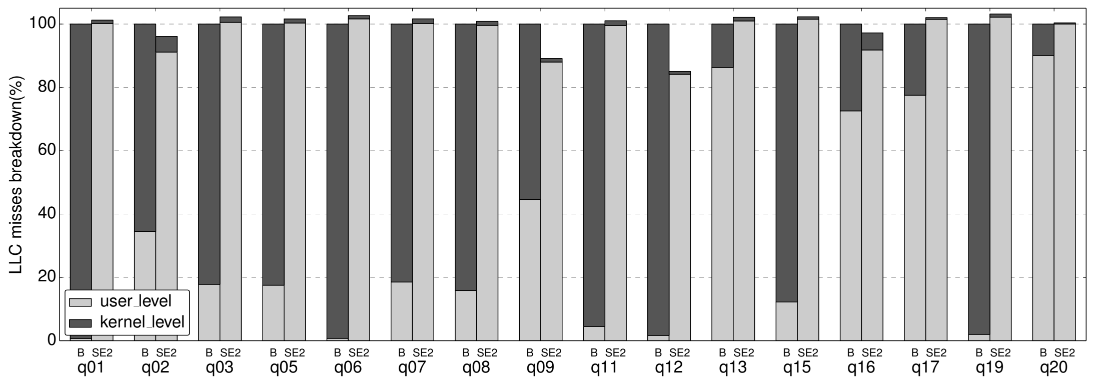


Figure 6.5: Last-level cache (LLC) misses breakdown — B = *pmfs_base95*, SE2 = *pmfs_se2*.

Figure 6.4 shows a classification of each cycle of execution as 'compute', if at least one instruction was committed during that cycle, or as 'stalled' otherwise. These categories are further broken down into user and kernel level cycles. Data is shown for *pmfs_base95* and *SE2*, normalized to the former. As can be seen, the *stalled_kernel* component correlates well with the kernel execution time shown in Figure 6.2, and this is the component that is reduced in *SE2* executions. Furthermore, reductions in the *stalled_kernel* component using SE2 are proportional to the time spent in DM operations shown in Figure 6.1. We observe that for most queries some of the savings from *stalled_kernel* shift to *stalled_user* since data needs to be brought close to the processing unit when it is needed for processing. There are some exceptions, i.e., Q11, Q15, and Q19, for which the simple prefetching scheme is able to mitigate this fact effectively.

Figure 6.5 shows a breakdown of user and kernel last-level cache (LLC) misses. Here, we can clearly see how the number of LLC misses remains quite constant when comparing *pmfs_base95* and *SE2*, but the misses shift from kernel level to user level. Moreover, in our experiments, we observe that user level misses have a more negative impact in terms of performance because they happen when the data is actually needed for processing, and a full LLC miss penalty is paid for each data element. On the other hand, when moving larger data blocks to an application buffer, optimized functions are employed and the LLC miss penalties can be overlapped.

6.1.4 Discussion

Experimental results show that there is a mismatch between the potential performance benefits shown in Figure 6.2 and the actual benefits in terms of wall-clock query execution time shown in Figure 6.3. Direct access to memory regions holding persistent data can provide significant benefits, but this data needs to be close to the processing units when it is needed. To this end, we employ simple software prefetching schemes that provide moderate average performance gains using our SE2 engine. However, carefully crafted ad-hoc software prefetching is challenging, and applications may not be designed in a way that makes it easy to hide long access latencies even with the use of prefetching, as happens with PostgreSQL. Moreover, such a solution is application and architecture dependent.

Additional software libraries and tools that aid programmability are needed in such systems. Such libraries could implement solutions like helper threads for prefetching particular data regions, effectively bringing data closer to the core (e.g., LLC) with minimal application interference. This approach would provide generic solutions for writing software that takes full advantage of the capabilities that NVM can offer. In Section 3.7, we explain a data prefetching library and its services along with three different thread mapping schemes. In the following subsection, we re-evaluate performance of SE2 when using library services to solve data readiness problem.

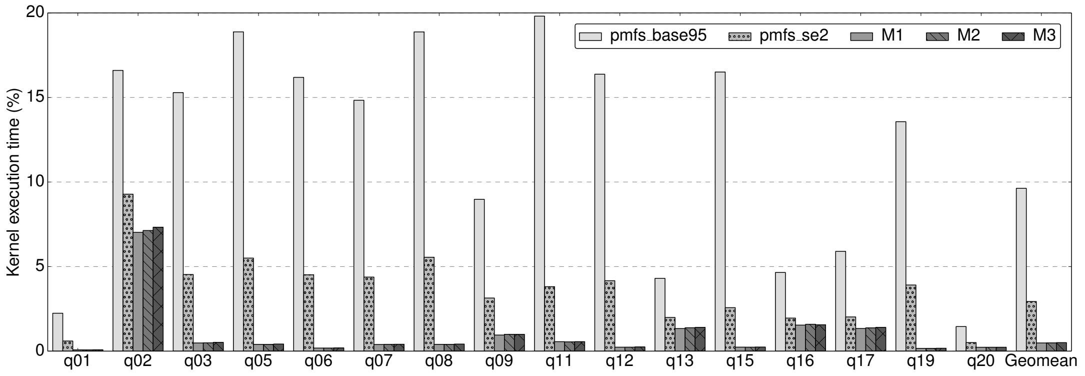


Figure 6.6: Percentage of kernel execution time for PostgreSQL thread.

6.1.5 Performance Evaluation of SE2 Coupled With Data Prefetching Library

For evaluation results presented in this section, SE2 no longer use the simple ad-hoc software prefetching scheme. Instead, it uses services of data prefetching library. However, the *pmfs_se2* system to which we compare does include the same ad-hoc prefetching used in the previous evaluation (Section 6.1.3). The test machine and methodology employed is the same as explained in Section 6.1.1, with the exception that hyper-threading (HT) is enabled for thread mapping schemes *M2* and *M3* (explained in Subsection 3.7.3).

6.1.5.1 Performance Impact on Kernel Execution Time

Figure 6.6 shows the percentage of kernel execution time of the PostgreSQL thread (computation thread) for each of the evaluated queries. As explained in Section 6.1.3, *SE2* only redirects the buffer pointer for file read operation from the local buffer cache to an NVM disk address that is within the address space of the PostgreSQL process. Hence, there is no data movement at the kernel level. As a result, the involvement of kernel in data movement and hence the average percentage of kernel execution time is already low in *pmfs_se2* as compared to *pmfs_base95*, reducing from 10% to 3%.

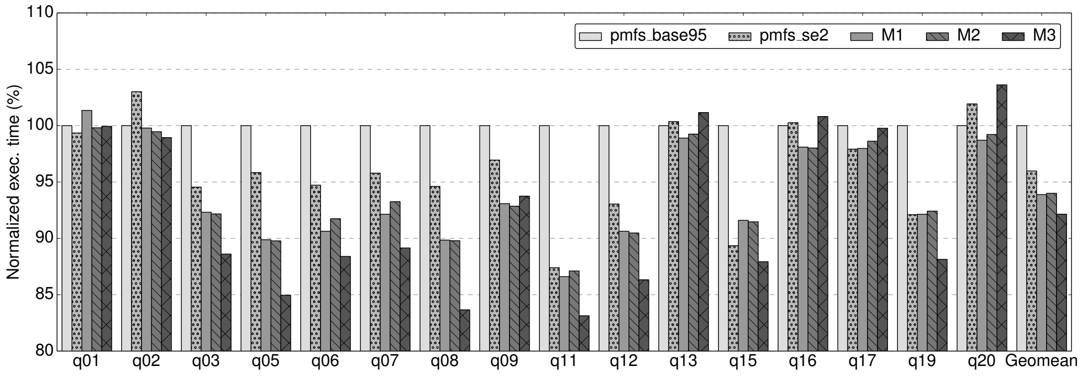


Figure 6.7: Wall-clock execution time normalized with respect to pmfs_base95.

When using `M1`, `M2`, and `M3` helper thread schemes, by offloading the prefetching of entire blocks of data to helper threads, we can further hide kernel execution time overheads for the PostgreSQL thread running a query. Helper threads are more effective in prefetching data than the ad-hoc scheme used in pmfs_se2. Therefore, the average percentage of kernel execution time further reduces from 3% for *pmfs_se2* to 0.5% in `M1`, `M2`, and `M3`.

There is no noticeable difference between the three thread mapping schemes in terms of percentage of kernel execution time. The reason is that all three mapping schemes place data blocks at least into the LLC, which is enough to hide kernel related events such as page faults.

6.1.5.2 Query Performance Improvement

Figure 6.7 shows the wall-clock query execution time normalized with respect to *pmfs_base95* for all queries. We can observe that for queries in which *pmfs_se2* obtained better performance, the new evaluated systems with our prefetch library overall obtain better execution times, especially for the `M3` thread mapping scheme. M3 shows noticeable performance improvements for queries where the sequential scan operation represents a significant fraction of the total database operations - i.e. Q03-Q12, Q15, and Q19, as shown in Figure 6.1. On the other hand, queries where sequential scan operation consume less time (i.e. Q01, Q02,

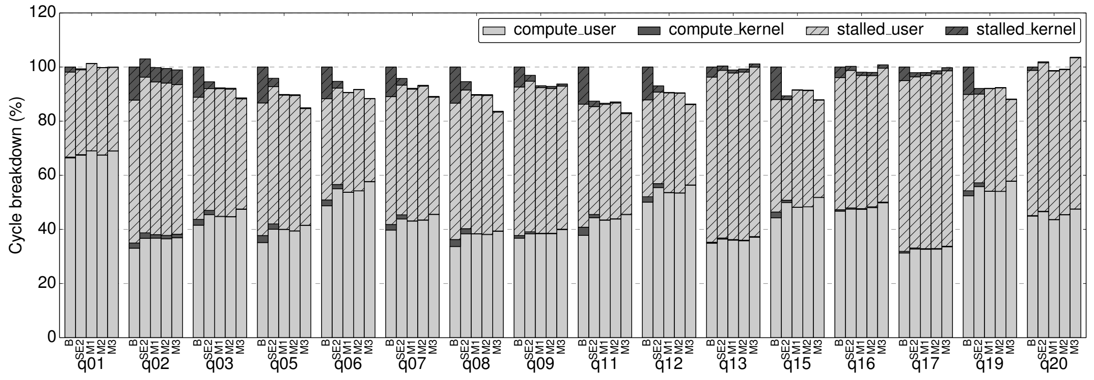


Figure 6.8: Execution time breakdown into compute and stall cycles for PostgreSQL thread, normalized with respect to pmfs_base95.

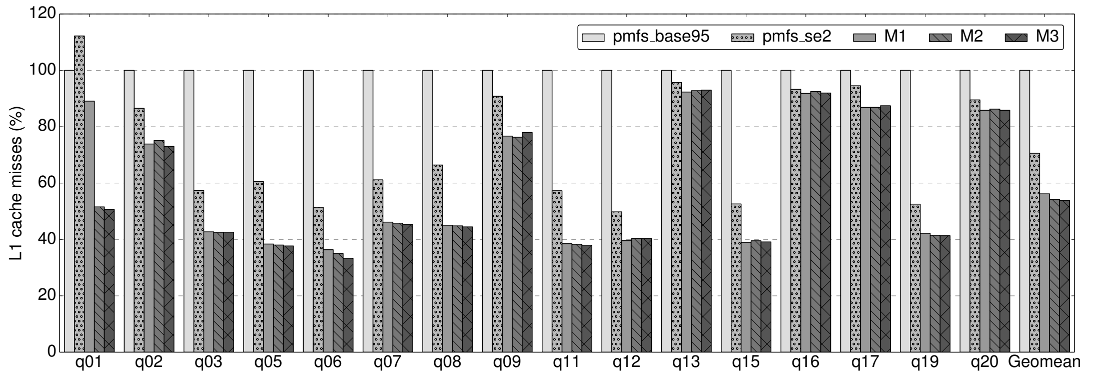


Figure 6.9: L1 cache misses for PostgreSQL thread normalized with respect to pmfs base95.

Q13, Q17, and Q20), show no performance improvement. On average, M3 obtains an 8% performance improvement over the baseline. `M1` and `M2` show up to 13% performance improvement (Q11), with an average of 6% when compared to pmfs_base95.

Query execution time is mostly affected by two factors: cache misses and competition for hardware resources between threads mapped on the same physical core. Reducing any of these two factors should lead to better query execution times. To understand the improvements seen in Figure 6.7, we provide insights in terms of compute and stalled core cycles and L1 cache misses for the PostgreSQL thread in Figure 6.8 and Figure 6.9, respectively.

Figure 6.8 shows an execution cycle break down into the stall and compute cycles for all evaluated queries, but just for the PostgreSQL thread. An execution cycle is classified as 'compute', if at least one instruction is committed during that cycle, or as 'stalled' otherwise. Both `M1` and `M2` generate similar results in terms of wall-clock execution time. In Figure 6.8, we can observe that both are able to reduce the kernel stall and compute components due to less kernel involvement in the main computation thread since the prefetching and kernel related events such as page faults are handled by the helper thread. However, the user level components are still very similar. This is because, as shown in Figure 6.9, the actual number of L1 cache misses is also similar despite helper threads prefetching at different levels of the memory hierarchy. We attribute this to hardware prefetchers being much more efficient once the data is already in the LLC. M3 shows better performance than M1 and M2 as it maps the helper-thread, which handles the page faults, on a core different than that of compute thread as shown in Figure 3.5. Nonetheless, Figure 6.9 shows a large L1 cache misses reduction when compared to both *pmfs_base95* and *pmfs_se2*, proving that the prefetching library is performing well in hiding them from the main computation thread.

We can see in Figure 6.8 that `M3`, besides being able to reduce the kernel components, also reduces the time spent in stalled user significantly, e.g., in Q03, Q05, Q06, Q12, and others. This is because of two factors: (i) prefetching into the L1 cache with our library is more timely than hardware prefetching, which might still be in-flight when the data is actually needed; and (ii) by using the two thread mapping approach we are able to reduce the overhead of the helper thread that is sharing the same physical core with the computation thread. Therefore, the first helper thread in `M3` brings the data into the LLC without interfering with the PostgreSQL thread that is running on a different physical core. While the second thread running on the same core brings the data into the private caches while incurring a lower overhead due to lower latency misses.

To support this explanation, Figure 6.10 shows the compute-stall cycle breakdown for the helper thread that shares the physical core with the computation PostgreSQL thread for `M2` and `M3` schemes. We can observe that for the queries

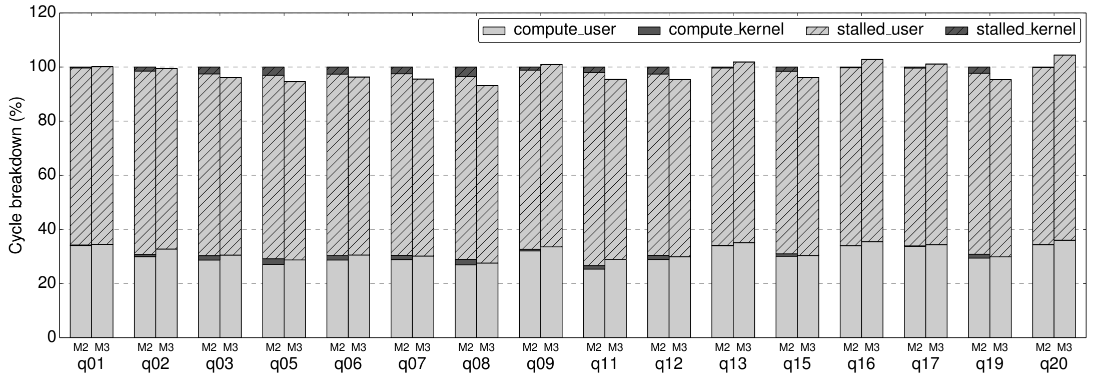


Figure 6.10: Execution time breakdown into compute and stall cycles for helper thread running on same physical core as PostgreSQL in M2 and M3. Execution time is normalized with respect to M2.

in which `M3` performs better (e.g., Q03, Q05, Q06, and Q12), the helper thread sharing the core is more lightweight. In particular, the helper thread in `M3` does not suffer from kernel noise since the kernel related events like page faults are sorted by the other helper thread running on a different core, leading to a load reduction in the computation core. Also, a reduction in user stalled cycles can be observed (e.g., Q11 and Q15) due to lower latency memory accesses.

We find that our library is able to help improve the performance with little programming effort. The ad-hoc prefetching scheme used in *pmfs_se2*, while simple, still required code analysis to place the prefetch instructions in different places in order to maximize the number of data blocks prefetched without stalling too much the computations. With our library the creation and mapping of threads is done only once, and the creation of jobs to be enqueued came as a natural fit in a single point of the source code, i.e., when the needed block is memory mapped.

6.2 Evaluation of Compiler Optimization

In this section, we evaluate the compiler optimization (detailed in Chapter 4) for reduction of memory bound stalls. We describe the experimental setup and benchmarks before presenting the evaluation results.

6.2.1 Experimental Setup

We use eleven image processing benchmarks for evaluation of the proposed compiler optimization. Table 6.2 provides a brief description of these benchmarks while the critical part of their source code is given in Appendix A.

Benchmarks P1-P4 and P6 perform image addition or subtraction on different number and sizes of input images. Although a very basic operation, image addition and subtraction is used as a step in other algorithms. Example of such algorithms includes usage of image differencing as a simple technique for change detection [168] providing a powerful interpretation of change in the tropical region and urban environment [169]. Image differencing is also used in mask mode radiography (for studying the propagation of contrast medium) and in motion-based segmentation [32]. Similarly, image addition is used in calculating the average face as a step in face recognition techniques based on eigenfaces [170]. The integral image technique is another algorithm which uses the pixel addition and is widely used in fields of computer vision and computer graphics such as texture mapping and face detection [171, 172].

Benchmark P5 and P8 perform filtering operation while P7 represents convolution. These operations are widely used for noise reduction, sharpening, edge detection and blurring of images [32, 173]. Benchmark P9 represents white balancing operation which is a required stage of image processing pipeline in modern digital cameras [174, 175]. Benchmark P10 consists of histogram generation which is used as an initial step in image enhancement applications [176, 177]. Benchmark P11 measures the degree of similarity for a given pixel in the first image with pixels in the second image at different disparities. The similarity measurement is an important step in all stereo matching algorithms [37, 178].

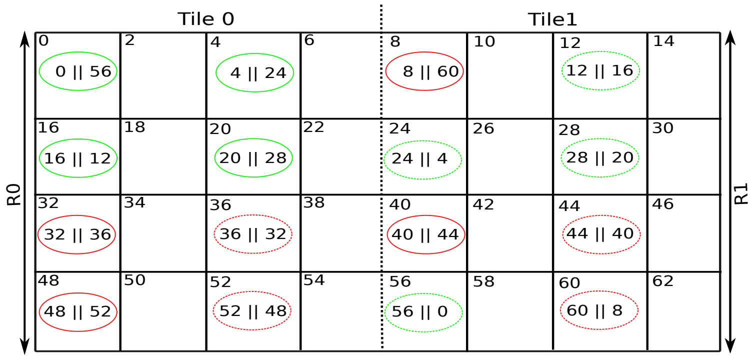
All benchmarks were executed on a Myriad 2 board with a single execution thread using both base compiler (BC) and affinity-aware compiler (AAC). For each benchmark, we measured the performance improvement as the percentage reduction in stall cycles and execution time when compared to the BC.

Table 6.2: Brief description of benchmarks.

Prog ID	Description	Number of Inputs	Total Input Size (KB)	Output Size (KB)
P1	Calculates the absolute difference of two input images.	2	1	1
P2	Performs addition of four input images.	4	20	20
P3	Performs addition of two input images.	2	1.875	1.875
P4	Same as P3 but performs addition of two input images based on a mask input.	3	0.5	0.5
P5	Calculates the output image as the scaled addition of five box filtered input images.	5	1.07	1.07
P6	Addition of two scaled images	2	20	20
P7	Image convolution using a 3x3 mask	1	3.75	1.25
P8	Sum of absolute difference using a 5x5 window	2	9.375	1.875
P9	Application of white balancing operation on a RGB image.	1	5.625	5.625
P10	Generation of histogram for the input image.	1	7.56	-
P11	Similarity measurement between two images.	2	0.625	4.03

```
     //i25=base_address+64
8)   LSU1.STO.16 i9 i25 -60
        || LSU0.STO.16 i10 i25 -40
7)   LSU1.STO.16 i28 i25 -8
        || LSU0.STO.16 i6 i25 -64
6)   LSU1.STO.16 i18 i25 -16
        || LSU0.STO.16 i29 i25 -12
5)   LSU1.STO.16 i16 i25 -24
        || LSU0.STO.16 i17 i25 -20
4)   LSU1.STO.16 i14 i25 -32
        || LSU0.STO.16 i15 i25 -28
3)   LSU1.STO.16 i12 i25 -44
        || LSU0.STO.16 i13 i25 -36
2)   LSU1.STO.16 i1 i25 -52
        || LSU0.STO.16 i11 i25 -48
1)   LSU1.STO.16 i27 i25 -4
        || LSU0.STO.16 i2 i25 -56
```

(a) A portion of BC-generated assembly code for Listing 6.1.



(b) Layout of relevant array elements in a CMX slice. Simultaneous memory accesses resulting in stalls are indicated by red colored ellipses.

Figure 6.11: Instruction schedule generated by BC for Listing 6.1.

6.2.2 Experimental Results for Source Code Annotation

Source code annotation is a time-consuming process and may need modifications at many places in the application source code. Therefore, we evaluate and demonstrate its working only for a single memory-intensive test program.

Listing 6.1 shows the source code of a simple test program. The program defines an array of short type of length SIZE. The `main()` function calls the `copyArray()` function which writes to every second element of the array. For performance evaluation, stalls and instruction execution cycles are measured for the *for loop* of `copyArray()` function.

A portion of assembly code generated by BC for Listing 6.1 is shown in Figure 6.11(a). In SHAVE's assembly, instructions scheduled in the same cycle are represented by placing || symbol among them. The syntax of a Store instruction is `LSU(0|1).STO.16 x,y,imm`, and it moves the data from the register `x` to memory. The memory address is calculated by using the content of the register `y` as the base address and `imm` as the displacement.

Listing 6.1: Source code of test program.

```
short A[SIZE];
void
__attribute__((noinline))  copyArray(short *A){
initializeTimers();
startTimers();
unsigned int i = 0;
for (i = 0; i < SIZE; i += 32){
  A[i] = 0;
  A[i + 2] = 2;
  A[i + 4] = 4;
  A[i + 6] = 6;
  A[i + 8] = 8;
  A[i + 10] = 10;
  A[i + 12] = 12;
  A[i + 14] = 14;
  A[i + 16] = 16;
  A[i + 18] = 18;
  A[i + 20] = 20;
  A[i + 22] = 22;
  A[i + 24] = 24;
  A[i + 26] = 26;
  A[i + 28] = 28;
  A[i + 30] = 30;
  }
  stopTimers();
}
int main(){
  copyArray(A);
  return 0;
}
```

In the assembly code of Figure 6.11(a) `i25` is a register containing the value of $base\_address + 64$. Numbers on the right of the assembly code show the clock cycles in which instructions are scheduled. For ease of discussion, we assume that *base_address* is zero. In cycle 1 of the assembly code, two Store instructions are scheduled simultaneously. One of them accesses address 60 (= 0+64-4), located in *Tile 1* and *R1* of the CMX slice as shown in Figure 6.11(b). The other paired instruction accesses the location at address 8 (= 0+64-56), also located in the same tile and the same region. As a result, the simultaneous scheduling of the two instructions incurs a stall cycle, as discussed in *Case B2* of Section 4.2. Same is true for cycle 4, 5, and 6.

On the other hand, the two store instructions scheduled in cycle 2 access addresses in different tiles and hence different regions. Specifically, one of them accesses the address 12 (=0+64-52), located in the *Tile 1* an *R1.* The other paired instruction accesses the address 16 (=0+64-48), located in the *Tile0* and *R0.* Therefore, simultaneous scheduling does not incur a stall cycle as discussed in *Case A* of Section 4.2. Same is true for cycle 3, 7, and 8. Note that the *optimization potential* for the given piece of assembly code shown in Figure 6.11(a) (and not for the whole program) is 4 as there are four cycles entertaining simultaneous memory accesses to the same region. The analysis of all (SIZE=2500) memory accesses show that the program of Listing 6.1 has an *optimization potential* of 234 cycles.

Listing 6.2 is functionally same as the Listing 6.1 but the source code is modified to append memory objects with their tile IDs. Line 1 and 2 define two macros, `Tile0` and `Tile1`, using the custom defined attribute i.e., `moviAttr`. As shown in line 13 to 28, instead of accessing the array elements through array index, each element is accessed by using a pointer carrying a tile ID of 0 or 1.

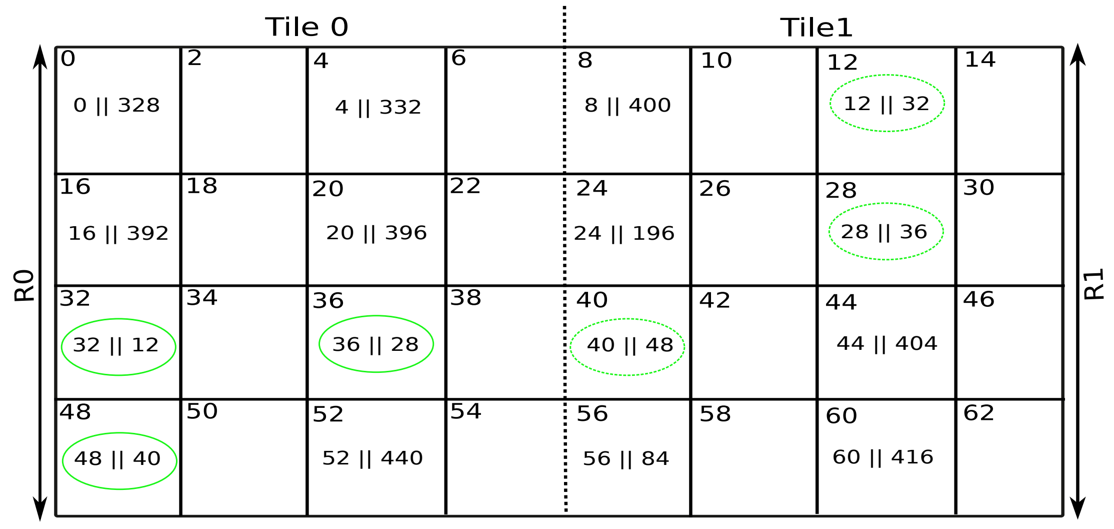
Figure 6.12(a) shows a portion of assembly code generated by AAC for the Listing 6.2 where register `i18` contains the value of $base\_address + 256$. For ease of discussion, we assume that *base_address* is zero. The layout of array elements in the CMX slice is shown in Figure 6.12(b). Assuming that the first element of the array is located at the address zero, the 16th element has an address of 32 (=16x2, where 2 is the size of each element) and the 18th element has an address of 36. Since line 21 and 22 of Listing 6.2 appends the same tile ID to 16th and 18th array elements, AAC does not schedule accesses to them in the same cycle.

Listing 6.2: Source code of test program with static affinity allocation.

```
#define Tile0 __attribute__((moviAttr(0)))
#define Tile1 __attribute__((moviAttr(1)))
short A[SIZE];
void __attribute__((noinline))  copyArray(short *A){
initializeTimers();
startTimers();
Tile0 short *temp1, *temp2;
Tile1 short *temp3, *temp4;
unsigned int i = 0;
for (i = 0; i < SIZE; i += 32){
  temp1 = A + i;   *temp1 = 0;
  temp2 = A + i + 2; *temp2 = 2;
  temp3 = A + i + 4; *temp3 = 4;
  temp4 = A + i + 6; *temp4 = 6;
  temp1 = A + i + 8; *temp1 = 8;
  temp2 = A + i + 10; *temp2 = 10;
  temp3 = A + i + 12; *temp3 = 12;
  temp4 = A + i + 14; *temp4 = 14;
  temp1 = A + i + 16; *temp1 = 16;
  temp2 = A + i + 18; *temp2 = 18;
  temp3 = A + i + 20; *temp3 = 20;
  temp4 = A + i + 22; *temp4 = 22;
  temp1 = A + i + 24; *temp1 = 24;
  temp2 = A + i + 26; *temp2 = 26;
  temp3 = A + i + 28; *temp3 = 28;
  temp4 = A + i + 30; *temp4 = 30;
  }
stopTimers();
}
int main(){
  copyArray(A);
  return 0;
}
```

```
      //i18=base_address+256             7) LSU1.STO.16 i10 i18 -232
                                                 || LSU0.STO.16 i9 i18 -60
13) LSU1.STO.16 i2 i18 72              6) LSU1.STO.16 i13 i18 -228
        || LSU0.STO.16 i6 i18 -256               || LSU0.STO.16 i15 i18 -220
12) LSU1.STO.16 i1 i18 76              5) LSU1.STO.16 i16 i18 -216
        || LSU0.STO.16 i9 i18 -252               || LSU0.STO.16 i29 i18 -208
11) LSU1.STO.16 i2 i18 -248            4) LSU1.STO.16 i17 i18 -212
        || LSU0.STO.16 i11 i18 144               || LSU0.STO.16 i12 i18 148
10) LSU1.STO.16 i1 i18 -244            3) LSU1.STO.16 i27 i18 184
        || LSU0.STO.16 i14 i18 -224              || LSU0.STO.16 i28 i18 -204
9)  LSU1.STO.16 i2 i18 136             2) LSU1.STO.16 i27 i18 -200
        || LSU0.STO.16 i11 i18 -240              || LSU0.STO.16 i12 i18 -172
8)  LSU1.STO.16 i1 i18 140             1) LSU1.STO.16 i26 i18 -196
        || LSU0.STO.16 i12 i18 -236              || LSU0.STO.16 i14 i18 160
```

(a) A portion of AAC-generated assembly code for Listing 6.2.



(b) Layout of relevant array elements in the CMX slice. All stalls resulting from simultaneous accesses in the same region (indicated by red colored ellipses) in the Figure 6.11 are removed by AAIS.

Figure 6.12: Instruction schedule generated by AAC for Listing 6.2.

As shown in Figure 6.12(a), address 32 (=256-224) is accessed simultaneously with address 12 (=256-244) in cycle 10 and both addresses are located in different tiles and hence different regions. This saves a stall cycle for accessing the 16th array element (located at address 32) as compared to the schedule generated by BC shown in Figure 6.11. In the same way, all other stall-generating simultaneous accesses in Figure 6.11 are avoided, resulting in significant reduction of stall cycles for the annotated version of the program.

Execution of the test program in Listing 6.1 on Myriad 2 board takes 937 clock cycles (i.e. 765 instruction cycles plus 172 stalls). On the other hand, the test program of Listing 6.2 takes 697 clock cycles (i.e. 641 instruction cycles plus 56 stalls). The remaining 56 stalls may belong to categories of bad speculation or core bound stalls. The difference of 240 cycles between the two execution times is very close to the *optimization potential* of 234 cycles for the test program in Listing 6.1. The significant improvement by 25.61% in execution time of the sample test program can be attributed to its memory intensive nature.

6.2.3 Experimental Results for Automated Affinity Analysis

In this subsection, we evaluate the classification of memory references using automated affinity analysis approach. We execute the benchmarks given in Table 6.2 on a Myriad 2 board using both BC and AAC.

Figure 6.13 shows the breakdown of the execution time of benchmarks into instruction and stall cycles. The breakdown is shown for both, BC and AAC. Benchmarks executed using AAC show a significant reduction in stall cycles as compared to the BC (e.g., P1, P2, P3, P4, P6, P9, and P10). The average reduction in stall cycles is by 69.83%.

Some benchmarks, such as P5 and P7, show a relatively lesser reduction in stall cycles. The difference in reduction of stall cycles across different benchmarks can be explained through their *optimization potential*. Figure 6.14 shows the

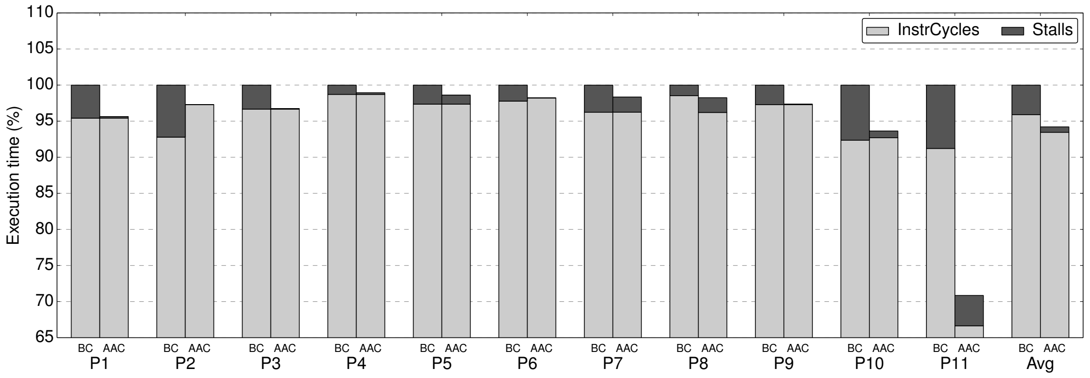


Figure 6.13: Breakdown of execution time into instruction and stall cycles for executions using BC and AAC.

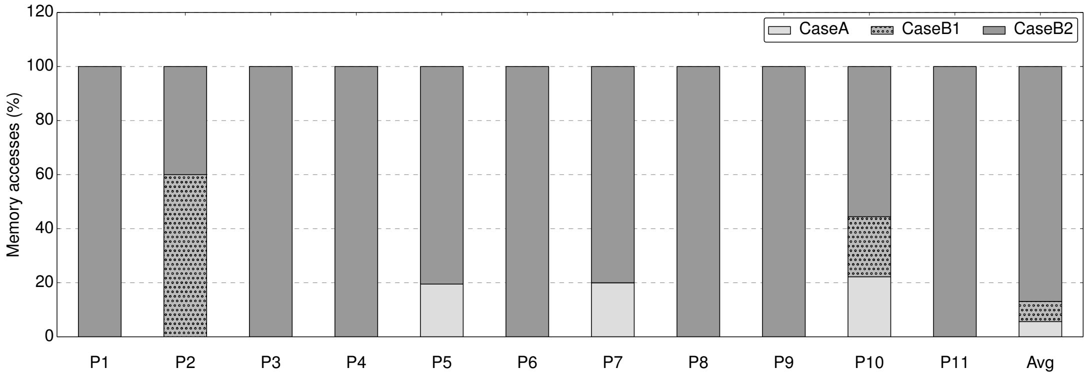


Figure 6.14: Breakdown of simultaneous memory access into three different cases. *Case A:* Memory objects belong to different tiles. *Case B:* Memory object belong to the same tile but different regions. *Case B_2:* Memory objects belong to the same region of the same tile.

breakdown of total requests for simultaneous memory access into three different cases. As explained in Section 4.2.3, BC and AAC generate the same schedule in *Case A*. In *Case B1*, AAC can possibly reduce the number of instruction cycles. However, AAC also reduces stalls in addition to a possible reduction in the number of instruction cycles in *Case B2*. In other words, higher the number of simultaneous memory requests belonging to *Case A* lower is the *Optimization Potential* of a benchmark. Figure 6.14 shows that both benchmarks, P5 and P7, have 20% of simultaneous memory requests belonging to *Case A*. Therefore, AAC achieves relatively lower reduction in stall cycles for P5 and P7 as compared to other benchmarks.

For most benchmarks (i.e., P1, P3, P4, P5, P6, P7, P9, and P10), the number of instruction cycles remains almost same when executed using BC and AAC. However, AAC execution of P2 increases the number of instruction cycles by 4.83%. The increase can be attributed to those simultaneously scheduled memory instructions in the BC-generated schedule which cannot be successfully serialized by AAC through scheduling them in the same cycle with another suitable instruction. As a result, new cycles need to be inserted for scheduling of such instructions (as explained through Schedule 3 of Table 4.1 in Section 4.2). On the other hand, AAC execution of P8 and P11 shows 2.34% and 24.57% reduction in instruction cycles, respectively, as compared to BC. This reduction can be attributed to those simultaneously scheduled memory instructions in the BC-generated schedule which are successfully serialized by AAC through scheduling them in the same cycle with another suitable instruction (as explained through the Schedule 1 of Table 4.1 in Section 4.2).

Fig. 6.13 shows that stall cycles make up to 8.77% (i.e. P11) of execution time and 4% on average for BC executions. Since our proposed optimization focuses only on the reduction of memory stalls, the execution time improves up to 30% (i.e. P11) with an average of 5.79%. However, the significant average reduction by 69% in stall cycles suggest that the proposed optimization can substantially improve execution time for memory intensive applications, as shown by a sample program in Section 6.2.2.

6.3 Evaluation of Selection Scheme for NoC Routing Algorithm

In this section, we present the evaluation results for adaptive routing scheme described in Chapter 5. We first describe the simulator and workload used for evaluation and then present experimental results.

6.3.1 Simulator and Benchmarks

We use Noxim, an NoC simulator developed using SystemC [179], for evaluation purpose. In Noxim, we can apply a given routing algorithm on a user defined traffic pattern specified as an input traffic trace file. A trace file must follow the format which is predefined by designers of Noxim. When a given algorithm is applied on an input traffic trace file, Noxim generates values for performance metrics and writes those values in an output trace file which can then be used to generate performance graphs.

We demonstrate the idea of switching routing algorithms during execution period of an application by using traffic trace files of SPARSE, FPPPP, H264-720p, and MPEG-4 decoder applications. Traffic trace file for SPARSE, FPPPP, and H264-720p is extracted from MCSL benchmark suite [180], while traffic trace file for MPEG-4 decoder is based on [181].

6.3.2 Experimental Results for SPARSE

MCSL benchmark suite provides traffic trace files for mapping of SPARSE application on mesh networks of different sizes. SPARSE is a medium size application with 96 tasks and 67 communication links. A communication link represents the flow of data from a task mapped on a source PE to a task mapped on a destination PE.

We simulate SPARSE on Noxim for simulation time of 100K cycles with the warm up period set to the initial 10K cycles. We select a 10x10 mesh NoC with the buffer size of four and fixed packet size of eight flits. Figure 6.15 shows the number of packets received after each interval, when XY and west-first routings (i.e. two candidate algorithms) are applied on the SPARSE traffic trace file. In this plot, simulation period of 100K is divided into 100 intervals each with the length of 1K cycles.

In Figure 6.15, we observe that for simulation period of 11K to 71K, number

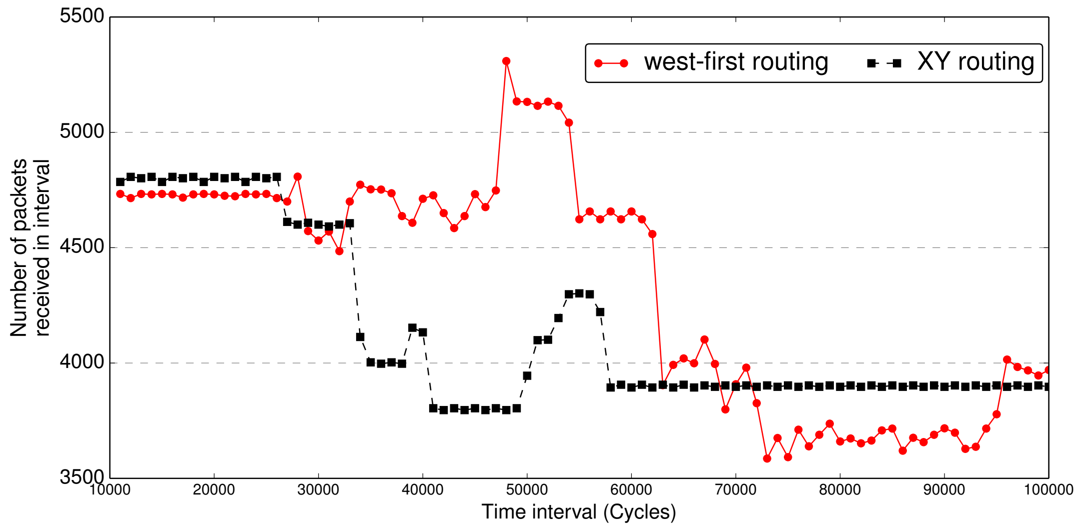


Figure 6.15: Number of received packets in the case of XY and west-first routings for SPARSE application.

of received packets in the case of west-first routing is more than or closer to that of XY routing algorithm. However, after 71K simulation cycles, number of received packets in the case of west-first routing is significantly lower than that of XY routing. This can be explained by the nature of traffic pattern formed by the active flows after the time point of 71K cycles. The graph in Figure 6.15 suggests that the application of west-first routing on the potential traffic pattern decreases number of received packets after the time point of 71K cycles, while XY routing performs better for that potential traffic pattern. Therefore we should apply west-first algorithm from the start of simulation to 71K cycles and then switch to XY routing algorithm to enhance total number of received packets.

Figure 6.16 shows the number of received packets in the case of static and dynamic selection between XY and west-first algorithms for SPARSE application. We observe that from the start of simulation to 71K cycles, number of received packets in the case of static selection is exactly the same as that for the west-first routing (see Figure 6.15) but after 71K cycles, number of received packets is higher than that for both XY and west-first routings.

To apply the dynamic selection between XY and west-first algorithms (using

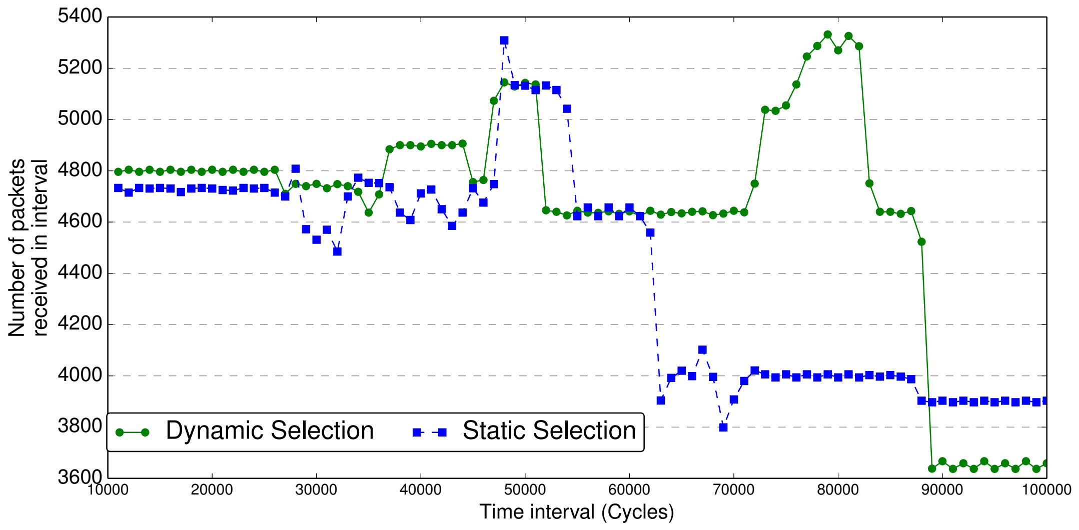


Figure 6.16: Number of received packets in the case of static and dynamic selection between XY and west-first routings for SPARSE application.

Algorithm 2), we first define congestion threshold points (CTPs) for a router based on the congestion level on that router. In our setting of Noxim, each router has five FIFOs, each one with the length of four. Hence, the total length of five buffers is 20. We define adaptivity list (AL) and CTPs as given below using the trial and error approach.

$CTP = \{4, 20\}$

$AL = \{XY, \textit{west-first}\}$

Figure 6.16 shows that the dynamic selection of two routing algorithms performs better as compared to the static selection. This is due to the fact that changes in the congestion level are monitored at the runtime and routing algorithms are selected accordingly.

Table 6.3 compares the throughput of individual application of XY and west-first routing algorithms with the static and dynamic selection between them. Throughput is calculated as a ratio of the total number of packets received in the steady state phase to duration of the phase. Static selection of XY and west-first algorithms improves throughput by 8.3% as compared to the XY routing

Table 6.3: Throughput comparison for SPARSE application.

Algorithm	**Number of Received Packets in Steady State**	**Throughput**
XY	39264	0.4908
west-first	42032	0.5254
Static Selection	42539	0.5317
Dynamic Selection	46086	0.5760

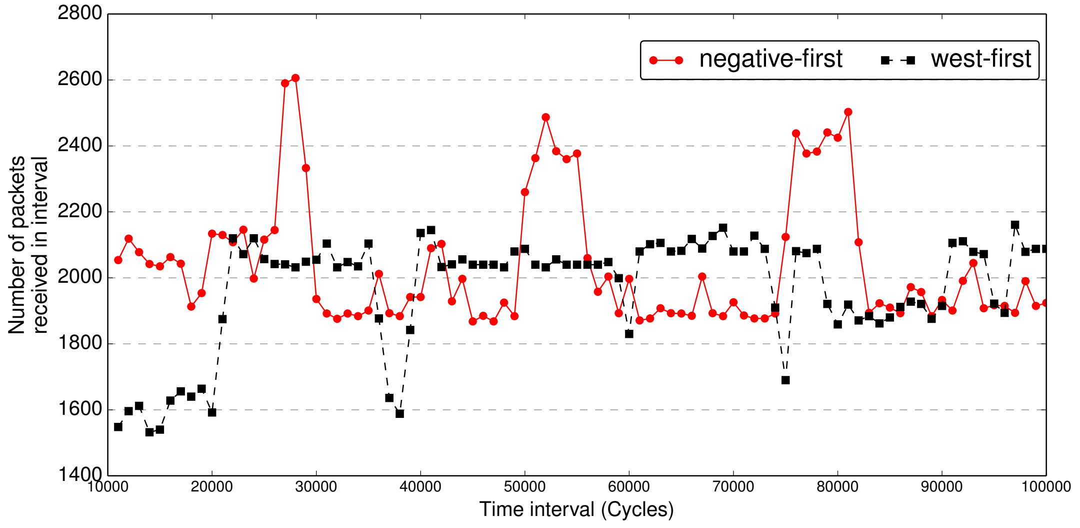


Figure 6.17: Number of received packets in the case of west-first and negative-first routings for MPEG-4 decoder application.

and 1.2% as compared to the west-first routing. For the dynamic selection case, throughput improves by 17.37% as compared to the XY routing and by 9.64% as compared to the west-first routing.

6.3.3 Experimental Results for MPEG-4 Decoder

We simulate the MPEG-4 decoder's task set on Noxim for a simulation time of 100K cycles with the warm up period set to the initial 10K cycles. We use a 4x4 mesh NoC with the buffer size of four and fixed packet size of eight flits.

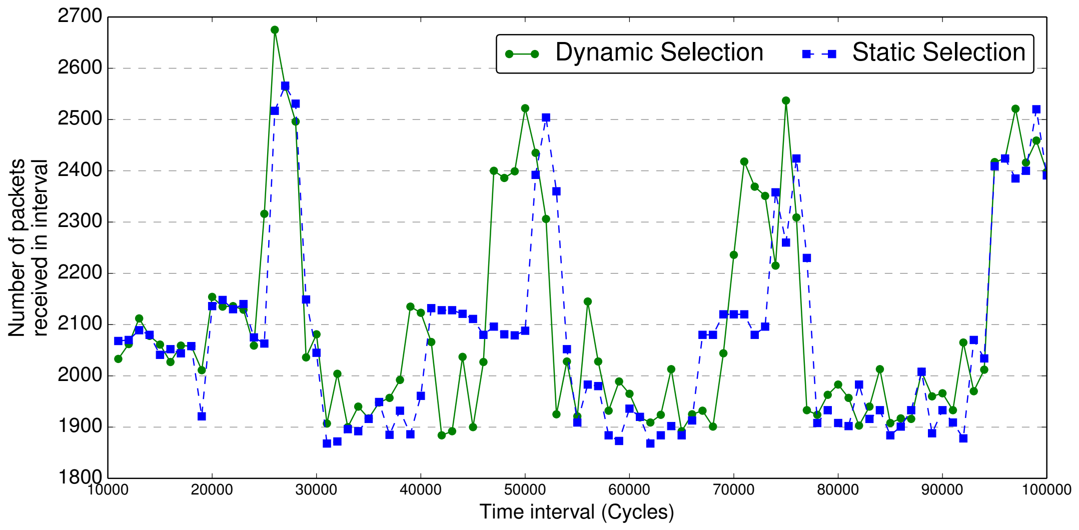


Figure 6.18: Number of received packets in the case of static and dynamic selection between west-first and negative-first routings for MPEG-4 decoder application.

Figure 6.17 shows the number of received packets when west-first and negative-first routing algorithms are applied individually on the MPEG-4 decoder application. It can be noted that the performance of these routing algorithms varies during simulation. For example, from the time point of 61K cycles to 74K cycles west-first routing performs better than the negative-first routing. On the other hand, after 74K simulation cycles, negative-first routing exhibits better performance.

It suggests that we can switch the routing algorithm during execution of the application, to gain a higher throughput. This is demonstrated in Figure 6.18, where graphs for number of received packets are drawn for the two cases i.e. static selection and dynamic selection between west-first and negative-first algorithms. For the dynamic selection between two algorithms, we define CTPs and AL as given below using the trial and error approach.

$CTP = \{4, 20\}$

$AL = \{\textit{west-first}, \textit{negative-first}\}$

Table 6.4: Throughput comparison for MPEG application.

Algorithm	**Number of Received Packets in Steady State**	**Throughput**
west-first	22142	0.2460
negative-first	22935	0.2548
Static Selection	23335	0.2592
Dynamic Selection	23636	0.2626

Table 6.4 compares the throughput of individual application of west-first and negative-first routing algorithms with the static and dynamic selection between them. Static selection between two routing strategies improves throughput by 5.39% and 1.74% as compared to the individual application of west-first and negative-first routing algorithms, respectively. Dynamic selection between two algorithms performs slightly better improving throughput by 6.74% and 3.05% as compared to west-first and negative-first routing algorithms, respectively.

6.3.4 Experimental Results for H264-720p

H264-720p is a H.264 video decoder application with the resolution of 720p. It consists of 2311 tasks communicating through 3461 links. We simulate the application on 16x16 mesh for 100K cycles with the warm up period set to the initial 10K cycles. We select the buffer size of four and fixed packet size of eight flits. Figure 6.19 shows the network throughput when west-first and north-last algorithms (i.e. two candidate algorithms) are applied for data routing between tasks of the application.

For static selection between routing algorithms, we set a switching point at 39K cycles. We apply west-first algorithm before and north-last algorithm after the switching point. For dynamic selection, we define CTPs and AL as given below.

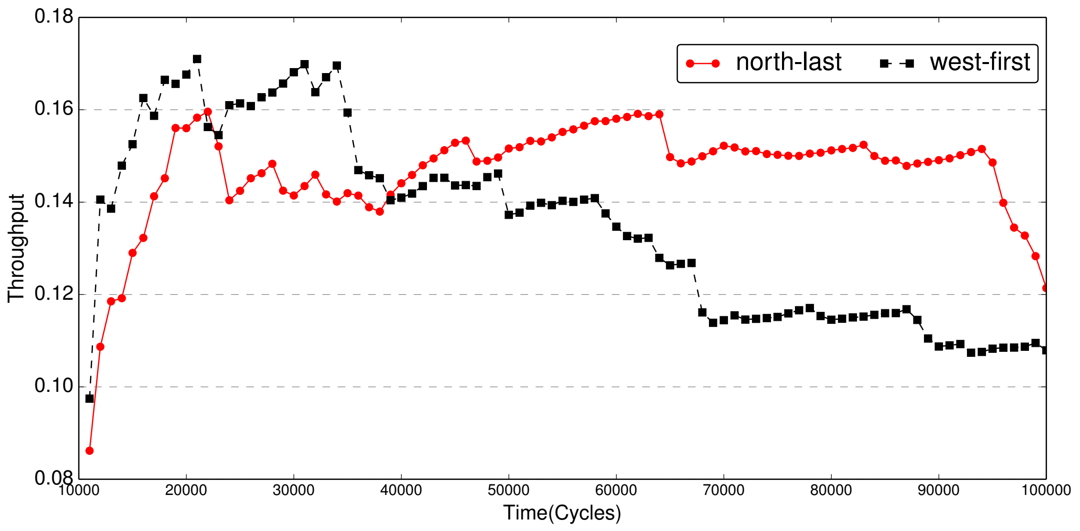


Figure 6.19: Network throughput in the case of west-first and north-last routings for H264-720p application.

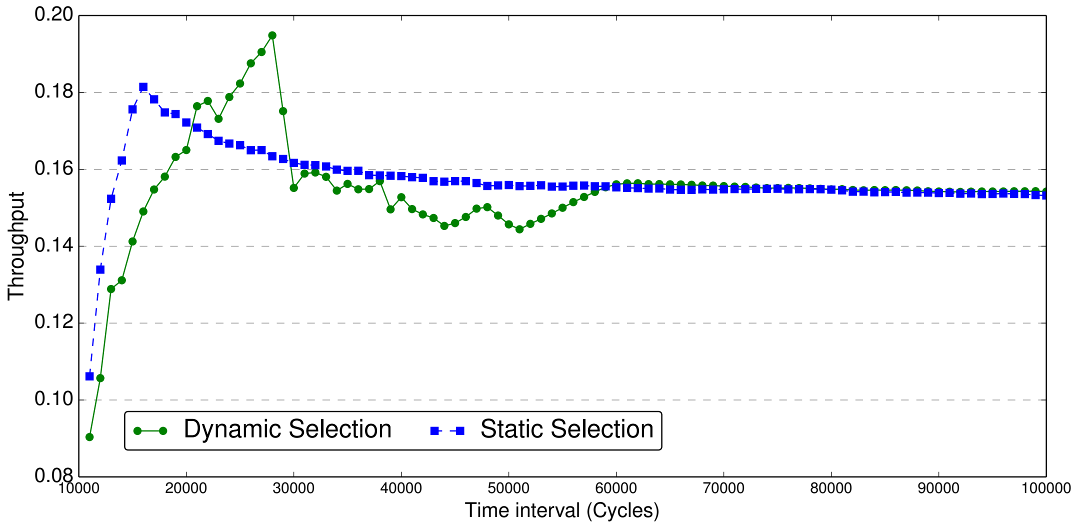


Figure 6.20: Network throughput in the case of static and dynamic selection of routing algorithms for H264-720p application.

$CTP = \{4, 20\}$

$AL = \{\textit{west-first}, \textit{north-last}\}$.

As shown in Figure 6.20, static selection improves network throughput by 14.74% and 5.54% as compared to the west-first and north-last algorithm, respectively. On the other hand, dynamic selection performs slightly better. It improves the network throughput by 16.75% and 7.39% as compared to the west-first and north-last algorithm, respectively.

6.3.5 Experimental Results for FPPPP

FPPPP is a chemical program performing multi-electron integral derivatives. It consists of 334 tasks with 1145 communication links among them. We simulate FPPPP on a 16x16 mesh for 100K cycles with the warm up period set to the initial 10K cycles. We select the buffer size of four and fixed packet size of eight flits. Figure 6.21 shows the network throughput for the application when west-first and north-last algorithms are applied.

For static selection between routing algorithms, we set a switching point at 42K cycles. We apply west-first and north-last algorithm before and after the switching point, respectively. For dynamic selection, we define CTPs and AL as given below.

$CTP = \{4, 20\}$

$AL = \{\textit{west-first}, \textit{north-last}\}$.

Static selection of routing algorithms performs same or worse than individual application of west-first and north-last algorithms. However, as shown in Figure 6.22, dynamic selection improves network throughput by 37.49% and 32.65% as compared to west-first and north-last algorithms.

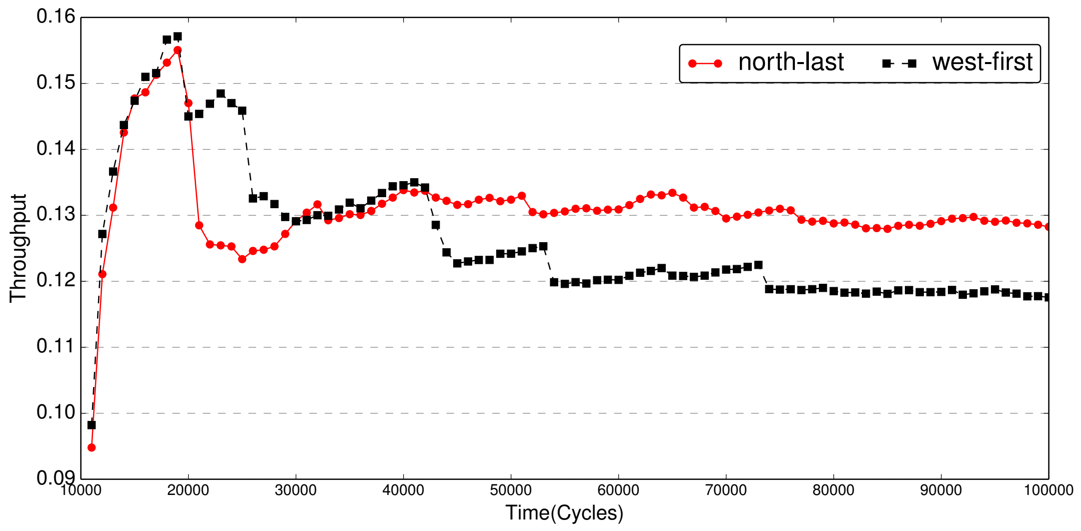


Figure 6.21: Network throughput in the case of west-first and north-last routings for FPPPP application.

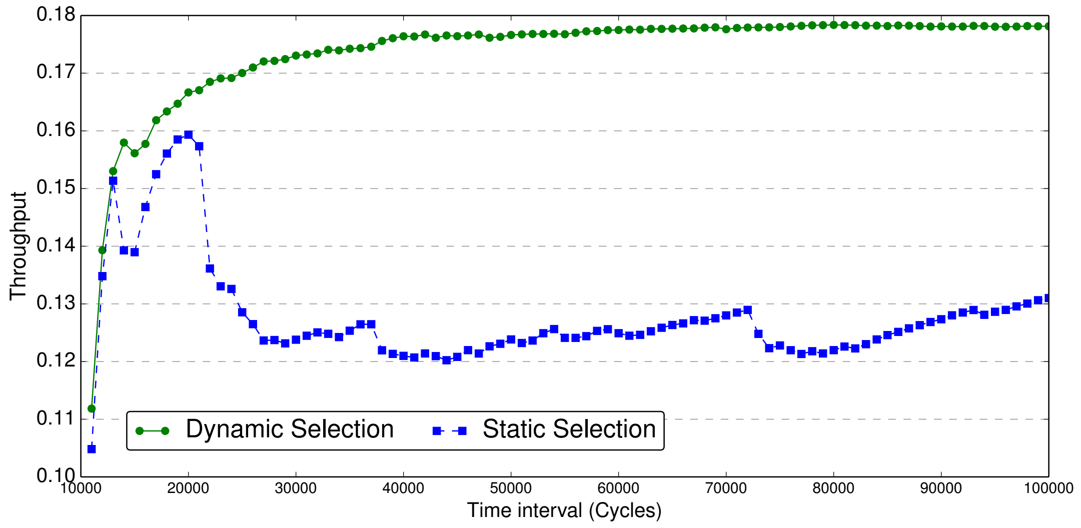


Figure 6.22: Network throughput in the case of static and dynamic selection of routing algorithms for FPPPP application.

Chapter 7

Conclusion

Our goal in this thesis is to improve performance of a computing system by i) reducing the performance gap between processing and memory components and ii) improving the throughput of network-on chip (NoC) in presence of changing traffic pattern of an application.

To achieve the first goal we make two proposals: First, we integrate improved and newly available non volatile memory as a storage device in the memory hierarchy of a computing system. Second, we reduce memory bound stalls by enabling a compiler to generate stall saving memory access schedule provided that underlying execution platform is equipped with necessary architectural features. To achieve the second goal, we propose a selection scheme aiming at selecting a routing algorithm which delivers higher throughput with changing traffic pattern.

More specifically, we address the integration of NVM storage in memory hierarchy of a computing system in the context of DBMS. We discuss the possible options to incorporate NVM into the memory hierarchy of a DBMS and conclude that, given the characteristics of NVM, a platform with a layer of DRAM where the disk is completely or partially replaced using NVM is a compelling scenario. Such an approach retains the programmability of current systems and allows direct access to the dataset stored in NVM. With this system configuration in mind we modify the PostgreSQL storage engine in two incremental steps - SE1 and SE2 - to better exploit the features offered by PMFS using memory mapped I/O.

Our evaluation shows that storing the dataset in NVM instead of disk for an unmodified version of PostgreSQL improves query execution time by up to 39%, with an average of 15%. Modifications to take advantage of NVM hardware improve the execution time by around 19% on average as compared to disk storage. However, the current design of database software proves to be a hurdle in maximizing the improvement. When comparing our baseline and *SE2* using PMFS, we achieve significant speedups of up to 13% in read-dominant queries, but moderate average improvements of 4% in query execution time.

We find that the limiting factor in achieving higher performance improvements is the fact that the data is not close to the processing units when it is needed for processing. This is a negative side effect of directly accessing data from NVM, rather than copying it into application buffers to make it accessible. This leads to long latency user-level cache misses eating up the improvement achieved by avoiding expensive data movement operations.

We develop a general purpose data prefetching library to mitigate this negative side effect. The library provides services to bring data into caches through user-level helper threads in a parallel fashion without stalling the application. Our library improves query execution time when compared to *disk_base95* by up to 54%, with an average of 23%. When compared to *pmfs_base95*, query execution time improves by up to 17%, with 8% average improvements.

Similar to the first proposal, our compiler optimization aims at reducing memory bound stalls by classifying the memory accesses. Our solution consists of two steps: affinity analysis and affinity-aware instruction scheduling. We implemented two different approaches for affinity analysis, namely, source code annotation and automated analysis.

While source code annotation approach facilitates more accurate prediction of the physical location of memory objects, it needs considerable effort by the programmer to modify the source code. On the other hand, automated analysis relieves a programmer from the burden of modifying the source code but employs certain assumptions about data placement. This may result with less accuracy in attaching affinity numbers to memory objects.

Experimental results show that by making the compiler aware of memory architecture and efficiently using the dual load-store units, memory stalls can be reduced significantly. Classification of memory references using source code annotation reduces stalls by 67.44% for a memory-intensive program, leading to 25.61% improvement in its execution time. On the other hand, automated analysis approach shows an average reduction of 69.83% in stall cycles with a modest improvement of 5.79% in execution time over a set of eleven different image processing benchmarks.

Our final contribution on selecting a routing algorithm with changing traffic pattern improves network throughput as compared to running a single algorithm over the entire execution period of an application. Next routing algorithm can be selected either in the static or dynamic way. However, dynamic selection of algorithms performs better as compared to the static one as algorithms are selected at the router level instead of NoC level and the selection is driven by the congestion information.

Bibliography

[1] "Computer | definition of computer by merriam-webster." `https://www.merriam-webster.com/dictionary/computer`. (Accessed on 12/31/2018).

[2] P. E. Ceruzzi *et al.*, *A history of modern computing.* MIT press, 2003.

[3] H. H. Goldstine and A. Goldstine, "The electronic numerical integrator and computer (eniac)," *Mathematical Tables and Other Aids to Computation*, vol. 2, no. 15, pp. 97–110, 1946.

[4] S. McCartney, *ENIAC: The triumphs and tragedies of the world's first computer.* Walker & Company, 1999.

[5] B. Gregg, *Systems performance: enterprise and the cloud.* Pearson Education, 2014.

[6] D. J. Lilja, *Measuring computer performance: a practitioner's guide.* Cambridge university press, 2005.

[7] M. M. Waldrop, "More than moore," *Nature*, vol. 530, no. 7589, pp. 144–148, 2016.

[8] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conventional microarchitectures," in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 248–259, ACM, 2000.

[9] C. Mack, "The multiple lives of moore's law," *IEEE Spectrum*, vol. 52, no. 4, pp. 31–31, 2015.

[10] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[11] D. J. Frank, "Power-constrained cmos scaling limits," *IBM Journal of Research and Development*, vol. 46, no. 2.3, pp. 235–244, 2002.

[12] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng, "Map-reduce for machine learning on multicore," in *Advances in neural information processing systems*, pp. 281–288, 2007.

[13] P. P. Gelsinger, "Microprocessors for the new millennium: Challenges, opportunities, and new frontiers," in *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*, pp. 22–25, IEEE, 2001.

[14] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, pp. 746–749, ACM, 2007.

[15] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[16] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of gpus and multicores," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, pp. 413–423, ACM, 2014.

[17] G. Teodoro, T. Kurc, G. Andrade, J. Kong, R. Ferreira, and J. Saltz, "Application performance analysis and efficient execution on systems with multi-core cpus, gpus and mics: a case study with microscopy image analysis," *The international journal of high performance computing applications*, vol. 31, no. 1, pp. 32–51, 2017.

[18] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[19] M. Pelcat, J. F. Nezan, J. Piat, J. Croizer, and S. Aridhi, "A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems," in *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, pp. 8–pages, 2009.

[20] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, p. 966, 2009.

[21] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics," *Journal of big data*, vol. 2, no. 1, p. 8, 2015.

[22] Y.-K. Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, "Signal processing on platforms with multiple cores: Part 1-overview and methodologies [from the guest editors]," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 24–25, 2009.

[23] L. Karam, I. AlKamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans, "Trends in multicore dsp platforms," *IEEE signal processing magazine*, vol. 26, no. 6, 2009.

[24] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "Pulp: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision," *Journal of Signal Processing Systems*, vol. 84, no. 3, pp. 339–354, 2016.

[25] O. Deniz, N. Vallez, J. L. Espinosa-Aranda, J. M. Rico-Saavedra, J. Parra-Patino, G. Bueno, D. Moloney, A. Dehghani, A. Dunne, A. Pagani, *et al.*, "Eyes of things," *Sensors*, vol. 17, no. 5, p. 1173, 2017.

[26] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, p. 23, 2007.

[27] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Cycle-accurate network on chip simulation with noxim," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 27, no. 1, p. 4, 2016.

[28] P. K. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *Journal of Systems Architecture*, vol. 59, no. 1, pp. 60–76, 2013.

[29] J. Hestness, B. Grot, and S. W. Keckler, "Netrace: dependency-driven trace-based network-on-chip simulation," in *Proceedings of the Third International Workshop on Network on Chip Architectures*, pp. 31–36, ACM, 2010.

[30] P. A. Gargini, "How to successfully overcome inflection points, or long live moore's law," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 51–62, 2017.

[31] A. Yazdanbakhsh, B. Thwaites, H. Esmaeilzadeh, G. Pekhimenko, O. Mutlu, and T. C. Mowry, "Mitigating the memory bottleneck with approximate load value prediction," *IEEE Design & Test*, vol. 33, no. 1, pp. 32–42, 2016.

[32] R. C. Gonzalez, R. E. Woods, S. L. Eddins, *et al.*, *Digital image processing using MATLAB.*, vol. 624. Pearson-Prentice-Hall Upper Saddle River, New Jersey, 2004.

[33] M. McDonnell, "Box-filtering techniques," *Computer Graphics and Image Processing*, vol. 17, no. 1, pp. 65–70, 1981.

[34] V. Podlozhnyuk, "Image convolution with cuda," *NVIDIA Corporation white paper, June*, vol. 2097, no. 3, 2007.

[35] H. Niitsuma and T. Maruyama, "Sum of absolute difference implementations for image processing on fpgas," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pp. 167–170, IEEE, 2010.

[36] S. Bianco, F. Gasparini, and R. Schettini, "Combining strategies for white balance," in *Digital photography III*, vol. 6502, p. 65020D, International Society for Optics and Photonics, 2007.

[37] H. Hirschmuller and D. Scharstein, “Evaluation of cost functions for stereo matching,” in *Computer Vision and Pattern Recognition, 2007. CVPR’07. IEEE Conference on*, pp. 1–8, IEEE, 2007.

[38] J. Reinders, “Vtune performance analyzer essentials,” *Intel Press*, 2005.

[39] T. Bjerregaard and S. Mahadevan, “A survey of research and practices of network-on-chip,” *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 1, 2006.

[40] Y. Wu, C. Lu, and Y. Chen, “A survey of routing algorithm for mesh network-on-chip,” *Frontiers of Computer Science*, vol. 10, no. 4, pp. 591–601, 2016.

[41] W. Zhang, L. Hou, J. Wang, S. Geng, and W. Wu, “Comparison research between xy and odd-even routing algorithm of a 2-dimension 3x3 mesh topology network-on-chip,” in *Global Congress on Intelligent Systems*, pp. 329–333, IEEE, 2009.

[42] C. J. Glass and L. M. Ni, “The turn model for adaptive routing,” *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2, pp. 278–287, 1992.

[43] M. Li, Q.-A. Zeng, and W.-B. Jone, “Dyxy: a proximity congestion-aware deadlock-free dynamic routing method for network on chip,” in *Proceedings of the 43rd annual Design Automation Conference*, pp. 849–852, ACM, 2006.

[44] J. Hu and R. Marculescu, “Dyad: smart routing for networks-on-chip,” in *Proceedings of the 41st annual Design Automation Conference*, pp. 260–263, ACM, 2004.

[45] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[46] N. R. Mahapatra and B. Venkatrao, “The processor-memory bottleneck: problems and solutions,” *Crossroads*, vol. 5, no. 3es, p. 2, 1999.

[47] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2016.

[48] S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches," *IEEE Transactions on Parallel and Distributed Systems*, p. 14, 2015.

[49] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, IEEE, 2015.

[50] J. P. Shim, M. Warkentin, J. F. Courtney, D. J. Power, R. Sharda, and C. Carlsson, "Past, present, and future of decision support technology," *Decision support systems*, vol. 33, no. 2, pp. 111–126, 2002.

[51] S. Chaudhuri, U. Dayal, and V. Ganti, "Database technology for decision support systems," *Computer*, vol. 34, no. 12, pp. 48–55, 2001.

[52] "Db-engines ranking - popularity ranking of database management systems." `https://db-engines.com/en/ranking`. (Accessed on 12/31/2018).

[53] T. P. P. Council, "Tpc-h benchmark specification," *Published at http://www. tcp. org/hspec. html*, vol. 21, pp. 592–603, 2008.

[54] N. U. Mustafa, A. Armejach, O. Ozturk, A. Cristal, and O. S. Unsal, "Implications of non-volatile memory as primary storage for database management systems," in *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on*, pp. 164–171, IEEE, 2016.

[55] N. U. Mustafa, M. J. O'Riordan, S. Rogers, and O. Ozturk, "Exploiting architectural features of a computer vision platform towards reducing memory stalls," *Journal of Real-Time Image Processing*, pp. 1–18, 2018.

[56] N. Ul Mustafa, O. Ozturk, and S. Niar, "Adaptive routing framework for network on chip architectures," in *Proceedings of the 2016 Workshop on*

Rapid Simulation and Performance Evaluation: Methods and Tools, p. 5, ACM, 2016.

[57] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, *et al.*, “Scuba: diving into data at facebook,” *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1057–1067, 2013.

[58] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, “Sap hana database: data management for modern business applications,” *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[59] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila, “Ibm soliddb: In-memory database optimized for extreme speed and availability.,” *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 14–20, 2013.

[60] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, *et al.*, “Blink: Not your father's database!,” in *International Workshop on Business Intelligence for the Real-Time Enterprise*, pp. 1–22, Springer, 2011.

[61] “Peloton – the self-driving database management system.” `https://pelotondb.io/`. (Accessed on 12/31/2018).

[62] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, *et al.*, “Self-driving database management systems.,” in *CIDR*, 2017.

[63] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou, “Sql server column store indexes,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 1177–1184, ACM, 2011.

[64] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, “Reducing the storage overhead of main-memory oltp databases with hybrid indexes,” in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1567–1581, ACM, 2016.

[65] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 29–41, ACM, 2011.

[66] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1243–1254, ACM, 2013.

[67] I. Lee and H. Y. Yeom, "A single phase distributed commit protocol for main memory database systems," in *Proceedings 16th International Parallel and Distributed Processing Symposium*, pp. 8–pp, IEEE, 2001.

[68] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor, "A prolegomenon on oltp database systems for non-volatile memory," *ADMS@ VLDB*, 2014.

[69] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, "Challenges and future directions for the scaling of dynamic random-access memory (dram)," *IBM Journal of Research and Development*, vol. 46, no. 2.3, pp. 187–212, 2002.

[70] A. Driskill-Smith, "Latest advances and future prospects of stt-ram," in *Non-Volatile Memories Workshop*, pp. 11–13, 2010.

[71] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[72] H. Plattner, "Sanssoucidb: An in-memory database for processing enterprise workloads.," in *BTW*, vol. 180, pp. 2–21, 2011.

[73] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in sap hana database: the end of a column store myth," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 731–742, ACM, 2012.

[74] J. Arulraj and A. Pavlo, “How to build a non-volatile memory database management system,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1753–1758, ACM, 2017.

[75] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable high performance main memory system using phase-change memory technology,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, 2009.

[76] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, *et al.*, “Sap hana adoption of non-volatile memory,” *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1754–1765, 2017.

[77] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, *et al.*, “Phase-change random access memory: A scalable technology,” *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[78] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, “The missing memristor found,” *nature*, vol. 453, no. 7191, p. 80, 2008.

[79] J. Arulraj, A. Pavlo, and S. R. Dulloor, “Let’s talk about storage & recovery methods for non-volatile memory database systems,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 707–722, ACM, 2015.

[80] J. Chang, P. Ranganathan, T. Mudge, D. Roberts, M. A. Shah, and K. T. Lim, “A limits study of benefits from nanostore-based future data-centric system architectures,” in *Proceedings of the 9th conference on Computing Frontiers*, pp. 33–42, ACM, 2012.

[81] R. Hagmann and M. D. Skeen, “Real-time query optimization in a decision support system,” Jan. 8 2002. US Patent 6,338,055.

[82] K. Wang, J. Alzate, and P. K. Amiri, “Low-power non-volatile spintronic memory: Stt-ram and beyond,” *Journal of Physics D: Applied Physics*, vol. 46, no. 7, p. 074003, 2013.

[83] T. Perez and C. A. De Rose, “Non-volatile memory: Emerging technologies and their impacts on memory systems,” *Porto Alegre*, 2010.

[84] J. Arulraj, M. Perron, and A. Pavlo, “Write-behind logging,” *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 337–348, 2016.

[85] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm, “Sofort: A hybrid scm-dram storage engine for fast data recovery,” in *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, p. 8, ACM, 2014.

[86] A. Chatzistergiou, M. Cintra, and S. D. Viglas, “Rewind: Recovery write-ahead system for in-memory non-volatile data-structures,” *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 497–508, 2015.

[87] Y. Huang, T. Liu, and C. J. Xue, “Register allocation for write activity minimization on non-volatile main memory for embedded systems,” *Journal of Systems Architecture*, vol. 58, no. 1, pp. 13–23, 2012.

[88] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis, “Instant recovery for main memory databases.,” in *CIDR*, 2015.

[89] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging locks for non-volatile memory consistency,” in *ACM SIGPLAN Notices*, vol. 49, pp. 433–452, ACM, 2014.

[90] S. D. Viglas, “Write-limited sorts and joins for persistent memory,” *Proceedings of the VLDB Endowment*, vol. 7, no. 5, pp. 413–424, 2014.

[91] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A durable and energy efficient main memory using phase change memory technology,” in *ACM SIGARCH computer architecture news*, vol. 37, pp. 14–23, ACM, 2009.

[92] “3d xpoint technology.” `https://www.micron.com/products/advanced-solutions/3d-xpoint-technology`. (Accessed on 12/31/2018).

[93] R. Intel, “Architecture instruction set extensions programming reference,” *Intel Corporation, Feb*, 2012.

[94] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the nvram era," *Proceedings of the VLDB Endowment*, vol. 7, no. 2, pp. 121–132, 2013.

[95] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-mt: a scalable storage manager for the multicore era," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pp. 24–35, ACM, 2009.

[96] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.

[97] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," 2011.

[98] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[99] S. Gao, J. Xu, B. He, B. Choi, and H. Hu, "Pcmlogging: reducing transaction logging overhead with pcm," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, pp. 2401–2404, ACM, 2011.

[100] Y. Son, H. Kang, H. Y. Yeom, and H. Han, "A log-structured buffer for database systems using non-volatile memory," in *Proceedings of the Symposium on Applied Computing*, pp. 880–886, ACM, 2017.

[101] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 393–404, 2011.

[102] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-execution," in *ACM SIGPLAN Notices*, vol. 37, pp. 159–170, ACM, 2002.

[103] C. Jung, D. Lim, J. Lee, and Y. Solihin, "Helper thread prefetching for loosely-coupled multiprocessor systems," in *| Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, p. 118, Ieee, 2006.

[104] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 306–317, IEEE Computer Society, 2001.

[105] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with opencv," *Communications of the ACM*, vol. 55, no. 6, pp. 61–69, 2012.

[106] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pp. 109–116, IEEE, 2011.

[107] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.

[108] J.-L. Chua, Y. C. Chang, and W. K. Lim, "A simple vision-based fall detection technique for indoor video surveillance," *Signal, Image and Video Processing*, vol. 9, no. 3, pp. 623–633, 2015.

[109] M. J. Gómez, F. García, D. Martín, A. de la Escalera, and J. M. Armingol, "Intelligent surveillance of indoor environments based on computer vision and 3d point cloud fusion," *Expert Systems with Applications*, vol. 42, no. 21, pp. 8156–8171, 2015.

[110] S. S. Rautaray and A. Agrawal, "Vision based hand gesture recognition for human computer interaction: a survey," *Artificial Intelligence Review*, vol. 43, no. 1, pp. 1–54, 2015.

[111] S. Suwajanakorn, I. Kemelmacher-Shlizerman, and S. M. Seitz, "Total moving face reconstruction," in *European Conference on Computer Vision*, pp. 796–812, Springer, 2014.

[112] N. Smolyanskiy, C. Huitema, L. Liang, and S. E. Anderson, "Real-time 3d face tracking based on active appearance model constrained by depth data," *Image and Vision Computing*, vol. 32, no. 11, pp. 860–869, 2014.

[113] Y. Bar, I. Diamant, L. Wolf, and H. Greenspan, "Deep learning with non-medical training used for chest pathology identification," in *Medical Imaging 2015: Computer-Aided Diagnosis*, vol. 9414, p. 94140V, International Society for Optics and Photonics, 2015.

[114] H. Greenspan, B. Van Ginneken, and R. M. Summers, "Guest editorial deep learning in medical imaging: Overview and future promise of an exciting new technique," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1153–1159, 2016.

[115] E. Ohn-Bar, A. Tawari, S. Martin, and M. M. Trivedi, "On surveillance for safety critical events: In-vehicle video networks for predictive driver assistance systems," *Computer Vision and Image Understanding*, vol. 134, pp. 130–140, 2015.

[116] D. K. Mandal, J. Sankaran, A. Gupta, K. Castille, S. Gondkar, S. Kamath, P. Sundar, and A. Phipps, "An embedded vision engine (eve) for automotive vision processing," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pp. 49–52, IEEE, 2014.

[117] B. Zhang, W. Huang, J. Li, C. Zhao, S. Fan, J. Wu, and C. Liu, "Principles, developments and applications of computer vision for external quality inspection of fruits and vegetables: A review," *Food Research International*, vol. 62, pp. 326–343, 2014.

[118] M. Aghbashlo, S. Hosseinpour, and M. Ghasemi-Varnamkhasti, "Computer vision technology for real-time food quality assurance during drying process," *Trends in food science & technology*, vol. 39, no. 1, pp. 76–84, 2014.

[119] J. Ma, D.-W. Sun, J.-H. Qu, D. Liu, H. Pu, W.-H. Gao, and X.-A. Zeng, "Applications of computer vision for assessing quality of agri-food products: a review of recent research advances," *Critical reviews in food science and nutrition*, vol. 56, no. 1, pp. 113–127, 2016.

[120] Y. Guo, Q. Zhuge, J. Hu, J. Yi, M. Qiu, and E. H.-M. Sha, “Data placement and duplication for embedded multicore systems with scratch pad memory,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 809–817, 2013.

[121] D. Wang, X. Du, L. Yin, C. Lin, H. Ma, W. Ren, H. Wang, X. Wang, S. Xie, L. Wang, *et al.*, “Mapu: A novel mathematical computing architecture,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 457–468, IEEE, 2016.

[122] Z. Lin, J. Sankaran, and T. Flanagan, “Empowering automotive vision with tis vision accelerationpac,” *TI White Paper*, 2013.

[123] “Technology | machine vision technology | movidius.” `https://www.movidius.com/technology`. (Accessed on 12/31/2018).

[124] E. Diken, M. J. O’Riordan, R. Jordans, L. Jozwiak, H. Corporaal, and D. Moloney, “Mixed-length simd code generation for vliw architectures with multiple native vector-widths,” in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pp. 181–188, IEEE, 2015.

[125] T. P. Chen, D. Budnikov, C. J. Hughes, and Y.-K. Chen, “Computer vision on multi-core processors: Articulated body tracking,” in *Multimedia and Expo, 2007 IEEE International Conference on*, pp. 1862–1865, IEEE, 2007.

[126] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe, “Myriad 2: Eye of the computational vision storm,” in *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pp. 1–18, IEEE, 2014.

[127] S. Thorarensen, “A back-end for the skepu skeleton programming library targeting the low-power multicore vision processor myriad 2,” 2016.

[128] A. Sethia, G. Dasika, T. Mudge, and S. Mahlke, “A customized processor for energy efficient scientific computing,” *IEEE Transactions on Computers*, vol. 61, no. 12, pp. 1711–1723, 2012.

[129] J. Cho, Y. Paek, and D. Whalley, "Efficient register and memory assignment for non-orthogonal architectures via graph coloring and mst algorithms," in *ACM SIGPLAN Notices*, vol. 37, pp. 130–138, ACM, 2002.

[130] R. Leupers and D. Kotte, "Variable partitioning for dual memory bank dsps," in *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, vol. 2, pp. 1121–1124, IEEE, 2001.

[131] M.-Y. Ko and S. S. Bhattacharyya, "Partitioning for dsp software synthesis," in *International Workshop on Software and Compilers for Embedded Systems*, pp. 344–358, Springer, 2003.

[132] A. Murray and B. Franke, "Fast source-level data assignment to dual memory banks," in *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pp. 43–52, ACM, 2008.

[133] V. Sipkova, "Efficient variable allocation to dual memory banks of dsps," in *International Workshop on Software and Compilers for Embedded Systems*, pp. 359–372, Springer, 2003.

[134] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Operation and data mapping for cgras with multi-bank memory," in *ACM Sigplan Notices*, vol. 45, pp. 17–26, ACM, 2010.

[135] W. Mi, X. Feng, J. Xue, and Y. Jia, "Software-hardware cooperative dram bank partitioning for chip multiprocessors," in *IFIP International Conference on Network and Parallel Computing*, pp. 329–343, Springer, 2010.

[136] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner, "Extending openmp for numa machines," in *Supercomputing, ACM/IEEE 2000 Conference*, pp. 48–48, IEEE, 2000.

[137] J. Antony, P. P. Janes, and A. P. Rendell, "Exploring thread and memory placement on numa architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport," in *International Conference on High-Performance Computing*, pp. 338–352, Springer, 2006.

[138] C. Lameter, “Numa (non-uniform memory access): An overview,” *Queue*, vol. 11, no. 7, p. 40, 2013.

[139] C. P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. G. Fernandes, “Memory affinity for hierarchical shared memory multiprocessors,” in *Computer Architecture and High Performance Computing, 2009. SBAC-PAD’09. 21st International Symposium on*, pp. 59–66, IEEE, 2009.

[140] A. Kleen and N. An, “Api for linux, suse labs, 2004.”

[141] H. Löf and S. Holmgren, “affinity-on-next-touch: increasing the performance of an industrial pde solver on a cc-numa system,” in *Proceedings of the 19th annual international conference on Supercomputing*, pp. 387–392, ACM, 2005.

[142] S. Lankes, B. Bierbaum, and T. Bemmerl, “Affinity-on-next-touch: an extension to the linux kernel for numa architectures,” in *International Conference on Parallel Processing and Applied Mathematics*, pp. 576–585, Springer, 2009.

[143] B. Goglin and N. Furmento, “Enabling high-performance memory migration for multithreaded applications on linux,” 2009.

[144] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, “Hexagon dsp: An architecture optimized for mobile multimedia and communications,” *IEEE Micro*, no. 2, pp. 34–43, 2014.

[145] K. Lee, S.-J. Lee, and H.-J. Yoo, “Low-power network-on-chip for high-performance soc design,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 2, pp. 148–160, 2006.

[146] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, “Routing table minimization for irregular mesh nocs,” in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE’07*, pp. 1–6, IEEE, 2007.

[147] C. Bobda and A. Ahmadinia, “Dynamic interconnection of reconfigurable modules on reconfigurable devices,” *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 443–451, 2005.

[148] T. Pionteck, R. Koch, and C. Albrecht, “Applying partial reconfiguration to networks-on-chips,” in *Field Programmable Logic and Applications, 2006. FPL’06. International Conference on*, pp. 1–6, IEEE, 2006.

[149] L. Devaux, S. B. Sassi, S. Pillement, D. Chillet, and D. Demigny, “Draft: Flexible interconnection network for dynamically reconfigurable architectures,” in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 435–438, IEEE, 2009.

[150] G. Haiyun, “Survey of dynamically reconfigurable network-on-chip,” in *Future Computer Sciences and Application (ICFCSA), 2011 International Conference on*, pp. 200–203, IEEE, 2011.

[151] H. Kimura, “Foedus: Oltp engine for a thousand cores and nvram,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 691–706, ACM, 2015.

[152] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, “System software for persistent memory,” in *Proceedings of the Ninth European Conference on Computer Systems*, p. 15, ACM, 2014.

[153] “Github - linux-pmfs/pmfs: Persistent memory file system.” `https://github.com/linux-pmfs/pmfs`. (Accessed on 12/31/2018).

[154] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger, “Track-aligned extents: Matching access patterns to disk drive characteristics.,” in *FAST*, vol. 2, pp. 259–274, 2002.

[155] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik, “Anti-caching: A new approach to database management system architecture,” *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1942–1953, 2013.

[156] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 265–276, IEEE Press, 2014.

[157] B. Momjian, *PostgreSQL: introduction and concepts*, vol. 192. Addison-Wesley New York, 2001.

[158] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 677–689, ACM, 2015.

[159] "Postgresql 9.0 documentation." `https://www.postgresql.org/files/documentation/pdf/9.0/postgresql-9.0-A4.pdf`. (Accessed on 12/31/2018).

[160] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 52–61, IEEE, 2001.

[161] "The llvm target-independent code generator — llvm 8 documentation." `https://llvm.org/docs/CodeGenerator.html#introduction-to-selectiondags`. (Accessed on 12/31/2018).

[162] M. Palesi and M. Daneshtalab, *Routing algorithms in networks-on-chip*. Springer, 2014.

[163] G.-M. Chiu, "The odd-even turn model for adaptive routing," *IEEE Transactions on parallel and distributed systems*, vol. 11, no. 7, pp. 729–738, 2000.

[164] A.-M. Rahmani, A. Afzali-Kusha, and M. Pedram, "A novel synthetic traffic pattern for power/performance analysis of network-on-chips using negative exponential distribution," *Journal of Low Power Electronics*, vol. 5, no. 3, pp. 396–405, 2009.

[165] G. B. Bezerra, S. Forrest, M. Moses, A. Davis, and P. Zarkesh-Ha, "Modeling noc traffic locality and energy consumption with rent's communication

probability distribution," in *Proceedings of the 12th ACM/IEEE international workshop on System level interconnect prediction*, pp. 3–8, ACM, 2010.

[166] L. Tedesco, A. Mello, D. Garibotti, N. Calazans, and F. Moraes, "Traffic generation and performance evaluation for mesh-based nocs," in *Proceedings of the 18th annual symposium on Integrated circuits and system design*, pp. 184–189, ACM, 2005.

[167] P. Gratz, B. Grot, and S. W. Keckler, "Regional congestion awareness for load balance in networks-on-chip," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 203–214, IEEE, 2008.

[168] M. Hussain, D. Chen, A. Cheng, H. Wei, and D. Stanley, "Change detection from remotely sensed images: From pixel-based to object-based approaches," *ISPRS Journal of photogrammetry and remote sensing*, vol. 80, pp. 91–106, 2013.

[169] S. Minu and A. Shetty, "A comparative study of image change detection algorithms in matlab," *Aquatic Procedia*, vol. 4, pp. 1366–1373, 2015.

[170] M. Turk and A. Pentland, "Eigenfaces for recognition," *Journal of cognitive neuroscience*, vol. 3, no. 1, pp. 71–86, 1991.

[171] F. C. Crow, "Summed-area tables for texture mapping," in *ACM SIGGRAPH computer graphics*, vol. 18, pp. 207–212, ACM, 1984.

[172] L. Jiang, H. Xie, and B. Pan, "Speeding up digital image correlation computation using the integral image technique," *Optics and Lasers in Engineering*, vol. 65, pp. 117–122, 2015.

[173] K. He, J. Sun, and X. Tang, "Guided image filtering," *IEEE transactions on pattern analysis & machine intelligence*, no. 6, pp. 1397–1409, 2013.

[174] R. Ramanath, W. E. Snyder, Y. Yoo, and M. S. Drew, "Color image processing pipeline," *IEEE Signal Processing Magazine*, vol. 22, no. 1, pp. 34–43, 2005.

[175] R. Lukac, "New framework for automatic white balancing of digital camera images," *Signal Processing*, vol. 88, no. 3, pp. 582–593, 2008.

[176] T. Arici, S. Dikbas, and Y. Altunbasak, "A histogram modification framework and its application for image contrast enhancement," *IEEE Transactions on image processing*, vol. 18, no. 9, pp. 1921–1935, 2009.

[177] J. Duan and G. Qiu, "Novel histogram processing for colour image enhancement," in *null*, pp. 55–58, IEEE, 2004.

[178] "Dspace@mit: A study of fast, robust stereo-matching algorithms." `https://dspace.mit.edu/handle/1721.1/61870#files-area`. (Accessed on 12/31/2018).

[179] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pp. 162–163, IEEE, 2015.

[180] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang, "A noc traffic suite based on real applications," in *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pp. 66–71, IEEE, 2011.

[181] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "Noc synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE transactions on parallel and distributed systems*, vol. 16, no. 2, pp. 113–129, 2005.

Appendix A

Critical part of source code for benchmarks

Listing A.1: Critical part of P1.

```
for (j = 0; j < width; j++){
  if (in_1[j] > in_2[j])
    out[0][j] = in_1[j] - in_2[j];
  else
    out[0][j] = in_2[j] - in_1[j];
  }
```

Listing A.2: Critical part of P2.

```
for (unsigned int k = 0; k < width; k++)
  {
  for (unsigned int disp = 0;
                  disp < disparities; disp++){
    out[k * disparities + disp]
    = (path0[k * disparities + disp]
    + path1[k * disparities + disp]
    + path2[k * disparities + disp]
        + path3[k * disparities + disp]) / 4;}
  }
```

Listing A.3: Critical part of P3.

```
for(i = 0; i < (int)width; i++){
add = src1[0][i] + src2[0][i];
if (add >= 255)
  add = 255.0f;
if (add <= 0)
  add = 0.0f;
dst[0][i] = (unsigned char)(add);
}
```

Listing A.4: Critical part of P4.

```
for(i = 0; i < (int)width; i++) {
if (mask[0][i] > 0){
  add = src1[0][i] + src2[0][i];
  if (add >= 255)
    add = 255.0f;
  if (add <= 0)
    add = 0.0f;
  dst[0][i] = (u8)(add);
  }
}
```

Listing A.5: Critical part of P5.

```
for (i = 0; i < width; i++){
 sum = 0;
 for (y = 0; y < 5; y++){
  for (x = -2; x <= 2; x++){
   sum += (lines[y][x]);
   }
   lines[y]++;
 }
*(*out+i)=(u8)(((half)(float)sum)*(half)0.04);
}
```

Listing A.6: Critical part of P6.

```
  for (col = 0; col < width; col++){
  for (disp = 0; disp < disparities; disp++){
  result = (alpha * disparityCost[col * disparities + disp]
+ beta * adCost[col * disparities + disp]) / normFactor;
  if (result > 255) result = 255;
    disparityCost[col * disparities + disp] = result;
  }
}
```

Listing A.7: Critical part of P7.

```
  for (i = 0; i < inWidth/3; i++){
  sum = 0.0f;
  for (x = 0; x < 3; x++)  {
    for (y = 0; y < 3; y++)
     sum += (short float)(lines[x][y - 1] * conv[x * 3 + y]);
    lines[x]+=3;
  }
  out[0][i] = (short float)(sum);
 }
```

Listing A.8: Critical part of P8.

```
  for (i = 0; i < width; i++){
  sum = 0;
  for (x = 0; x < 5; x++){
    for (y = 0; y < 5; y++){
      diff = lines1[x][y - 2] - lines2[x][y - 2];
      if(diff < 0)
        diff = 0 - diff;
      sum += diff; }
    lines1[x]++;
    lines2[x]++; }
  if (sum >= 255)
    sum = 255;
  out[0][i] = (unsigned char)(sum);}
```

Listing A.9: Critical part of P9.

```
  for (i = 0; i < (int)width; i++){
  r = ((unsigned int)rIn[i] * (unsigned int)awbCoef[0]) >>15;
  g = ((unsigned int)gIn[i] * (unsigned int)awbCoef[1]) >>15;
  b = ((unsigned int)bIn[i] * (unsigned int)awbCoef[2]) >>15;

  rOut[i] = (unsigned short) (r > clamp[0] ? clamp[0] : r);
  gOut[i] = (unsigned short) (g > clamp[0] ? clamp[0] : g);
  bOut[i] = (unsigned short) (b > clamp[0] ? clamp[0] : b);
}
```

Listing A.10: Critical part of P10.

```
  for (i = 0; i < width; i+=4){
 int out1 = *piHist1;
 int out2 = *piHist2;
 int out3 = *piHist3;
 int out4 = *piHist4;

 *piHist1 = out1+1; *piHist2 = out2+1;
 *piHist3 = out3+1; *piHist4 = out4+1;

 piHist1 = hist1 + index1;
 piHist2 = hist2 + index2;
 piHist3 = hist3 + index3;
 piHist4 = hist4 + index4;

 index1 = (unsigned int)in_line[i + 8];
 index2 = (unsigned int)in_line[i + 9];
 index3 = (unsigned int)in_line[i + 10];
 index4 = (unsigned int)in_line[i + 11];
}
```

Listing A.11: Critical part of P11.

```
for (int positionL = 0;
        positionL < (width&0xfffffffc);
        positionL++) {

    unsigned int in1L = in1[positionL];
    unsigned int input[DISPARITIES];

    #pragma unroll DISPARITIES
    for(int i = DISPARITIES-4; i >= 0; i-=4)
      *((uint4 *) &input[i])
      = ((uint4 *) &in2[positionL-i-3])->s3210;

    #pragma unroll DISPARITIES
    for (unsigned int indexR = 0;
    indexR < DISPARITIES; indexR++) {
      unsigned int resultXOR = in1L ^ input[indexR];
      std::bitset<32> bits = resultXOR;
      out[positionL * DISPARITIES + indexR]
      = (unsigned char) bits.count();
    }
  }
```