UNIVERSITY OF CALIFORNIA,
IRVINE

Runtime Memory Management in Many-core Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Hossein Tajik

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Tony Givargis
Professor Alex Nicolau

2016

DEDICATION

To my mother, father, and sister for all their support, kindness, and love.
In memory of Dina Radjabalipour, who will be in our thoughts forever.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I would like to express my gratitude to my advisor Professor Nikil Dutt. Throughout these years, he was my role model and his enthusiasm and passion was the main driver for me to do a better research. Without his knowledge and guidance, it was impossible for me to finish my PhD and I learned a lot from him. He is a great person and wonderful mentor and I am very thankful for his patience toward me.

I also want to thank my previous mentors who helped me reach this point in my life. Dr. Houman Homayoun was a huge help for me during the first year of my PhD. Prof. Alireza Ejlai was my advisor during my Masters program and helped me a lot to learn about Embedded Systems. Dr. Ramin Halavati was the first one who taught me how to research. Reza Sadigh was my first computer programming teacher and I became interested in computer science because of him. Aydin Khatamnejad helped me to be a better programmer. I am very grateful to have these wonderful people in my life.

I want to thank UC Irvine graduate division and NSF Variability Expedition (Grant Number CCF-1029783) for providing funding opportunities during my PhD program. I also thank ACM for permission to include Chapter Two and Four of my dissertation, which were originally published in ACM TECS journal and ESTIMedia symposium respectively.

I was very lucky to be in DRG and spend time with many great people. Majid Namaki Shoushtari was a true friend and helped me a lot during hard times and Bryan Donyanavard helped me to shape my research. I am also very thankful for Kasra Moazzemi, Abbas Banaiyan, Luis (Danny) Bathen, codrut stancu, Janmartin Jahn, Tiago Rogerio Muck, JurnGyu Park, Roger Chen-Ying Hsieh, Hamid Nejatollahi, Trent Lo, Gustavo Girao, Santanu Sarma, Dr. Amir Rahmani, Prof. MyungKeun Yoon, , Prof. Antonio Augusto Frhlich, Jun Y Shin, Kazuyuki Tanimura, Juan Gonzalez, Prof. Alfonso Avila, Prof. Yukio Mitsuyama, Prof. Gu-Min Jeong, and Dr. Yuko Hara-Azumi.

There were many people who brought laughter and joy to my life. I am very grateful for them, especially I want to thank Mojtaba Torkjazi, Shahab Yassemi, Amir Gholamipour, Maryam Balouch, Hamed Youssefpour, Andy Jackson, Andrea Thorstensen, Hossein Tajari, Noah Monavvary, Mohammad Khorramzadeh, Pouria Pirzade, Arash Karami, Reza Baghaei, Sholeh Forouzan, Sonja Lind, Alandi Bates, Hamid Hezari, and Hessam Kooti.

I would like to express my gratitude to Prof. Alex Nicolau and Prof. Tony Givargis who helped me to revise and improve my work during the past four years. They were great sources of knowledge and experience for me. I would like to thank Prof. Hoyoung Hwang and Prof. Sung-Soo Lim for their friendliness, knowledge, and generosity. I am also very grateful for Prof. Jorge Henkel for all his advices which improved my research. I also want to thank Prof. Michael Dillencourt who is a great man and provided a lot of teaching opportunities for me. Last but not the least, I want to thank Melanie Sanders, Melanie Kilian, and Grace Wu for providing a great atmosphere in ICS department and CECS.

CURRICULUM VITAE

Hossein Tajik

EDUCATION

Doctor of Philosophy in Computer Science — **2016**
University of California, Irvine — *Irvine, California*

Master of Science in Computer Engineering — **2010**
Sharif University of Technology — *Tehran, Iran*

Bachelor of Science in Computer Engineering — **2007**
Sharif University of Technology — *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2011–2016**
University of California, Irvine — *Irvine, California*

Graduate Research Assistant — **2009–2010**
Sharif University of Technology — *Tehran, Iran*

Undergraduate Research Assistant — **2004–2006**
Sharif University of Technology — *Tehran, Iran*

TEACHING EXPERIENCE

Teaching Assistant — **2013–2016**
University of California, Irvine — *Irvine, California*

Teaching Assistant — **2009–2010**
Sharif University of TEchnology — *Tehran, Iran*

REFEREED JOURNAL PUBLICATIONS

SPMPool: Runtime SPM Management for Memory-intensive Applications in Embedded Many-Cores **2016**
ACM Transaction on Embedded Computing Systems

Automatic Management of Software Programmable Memories in Manycore Architectures **2016**
IET Computers & Digital Techniques

REFEREED CONFERENCE PUBLICATIONS

On Detecting and Using Memory Phases in Multimedia Systems **2016**
Estimedia

Orchestrated application quality and energy storage management in solar-powered embedded systems **2015**
ISQED

VAWOM: Temperature and process variation aware WearOut Management in 3D multicore architecture **2013**
Design Automation Conference (DAC)

A Novel Approach to Very Fast Isolated Word Speech Recognition **2006**
International Conference on Pattern Recognition

A novel noise immune, fuzzy approach to speaker independent, isolated word speech recognition **2006**
World Automation Congress

ABSTRACT OF THE DISSERTATION

Runtime Memory Management in Many-core Systems

By

Hossein Tajik

Doctor of Philosophy in Computer Science

University of California, Irvine, 2016

Professor Nikil Dutt, Chair

With the number of cores on a chip continuing to increase, we are moving towards an era where many-core platforms will soon be ubiquitous. Efficient use of tens to hundreds of cores on a chip and their memory resources comes with unique challenges. Some of these major challenges include: 1) Data Coherency – the need for coherency protocol and its induced overhead poses a major obstacle for scalability of many-core platforms. 2) Memory requirement variation – concurrently running applications on a many-core platform have variable and different memory requirements, not only across different applications, but also within a single application; in this dynamic scenario, static analysis may not suffice to capture dynamic behaviors. 3) Scalability – inefficiency of a central management makes distributed management a necessity for many-core platforms.

To address all these issues, this dissertation proposes a comprehensive approach to manage available memory resources in many-core platforms equipped with Software Programmable Memories (SPMs). The main contributions of this dissertation are: 1) We introduce SPM-Pool: a scalable platform for sharing Software Programmable Memories. The SPMPool approach exploits underutilized memory resources by dynamically sharing SPM resources between applications running on different cores and adapts to the overall memory requirements of multiple applications that are concurrently executing on the many-core platform.

2) We propose different central and distributed management schemes for SPMPool and study the efficiency of auction-based mechanisms in solving the memory mapping problem. We also introduce a distributed auction-based scheme to manage the memory resources of platforms without central coordination. 3) We introduce offline and online memory phase detection methods in order to increase the adaptivity of memory management to the temporal changes in memory requirements of a single application. We also use memory phasic information to relax the need for static analysis of applications.

We implemented a Java and Python based simulator for many-core platforms to investigate the efficacy of the proposed methods in this dissertation. The runtime memory management schemes proposed here enable better performance, power, and scalability for many-core systems.

Chapter 1

Introduction

1.1 Emerging Many-core Systems

Moore's law has been the driving force for the semiconductor industry in the past decades. Every technology generation tends to double the number of transistors on each die by reducing the feature size. Smaller transistors can be switched on and off faster which gives a potential for higher frequencies. Improvements in integrating different technologies in a single die has provided the ability to have a full System On a Chip (SOC). This ever-increasing resources on a chip has provided a unique opportunity for development of more powerful computer systems.

Providing faster and smaller transistors by each technology generation came with the cost of increased power density – more activity in smaller area. Increase in power density caused higher temperatures and reaching thermal limits of ICs, known as *Power Wall*, stalled the frequency growth of computer systems. Figure 1.1 shows the microprocessor trend in the past decades. While transistor count has been continued to increase in each generation, frequency has been almost the same for the past 15 years.

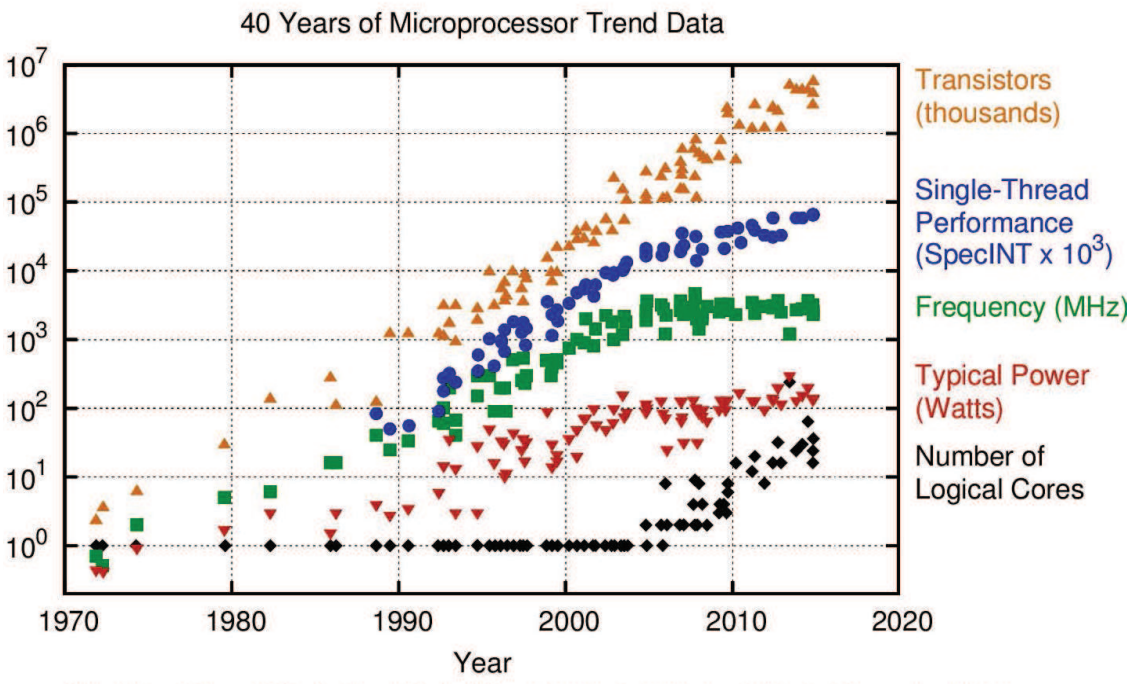


Figure 1.1: 40 years of microprocessor trend [102].

Inability to find enough Instruction Level Parallelism –*ILP Wall*– was another major obstacle [33, 61] that alongside with *Power Wall* put an end to development of Superscalar uni-core processors and were driving forces for multi-core processors. In these processors, better utilization of hardware resources are obtained by multi-threading.

IBM Power4 [128] was the first commercial processor to integrate two cores on a same die in 2001. From that time on, multi-core processors have been viable solutions for effective use of huge numbers of transistors on a chip. With increase in the number of cores on a chip, we are moving to the era where many-core platforms – platforms with hundreds of cores – will be common. Table 1.1 shows the trend of increase in the number of cores in mobile devices and micro-servers – used in data centers, predicted by ITRS [71]. Based on this prediction, it's expected to have more than 30 AP (Application Processor) cores and 200 GPU cores in mobile devices, and about 100 cores in micro-servers in less than 10 years, with the number of cores usually constrained by the power budget.

Table 1.1: ITRS Mobile Devices Trends [71][23]

	Year	2015	2017	2019	2021	2023	2025
Mobile Devices	Number of AP cores	4	9	18	18	28	36
	Number of GPU cores	6	19	49	69	141	247
	Max frequency of any component in system (GHz)	2.7	2.9	3.2	3.4	3.7	4
	Bandwidth between AP and Main memory (Gb/s)	25.6	34.8	52.6	57.3	61.9	61.9
Severs	Server Units/Rack	40	40	40	40	40	40
	Number of Cores/socket	18	29	47	59	74	93
	Main Memory/Single Server Unit (GB)	32	45	64	76	91	108

Mobile devices were invented as portable telephones with the ability to receive and make phone calls. During the past decades they evolved to very powerful computer systems. The number of smartphones, tables, and other mobile devices users has already outnumbered PC users [123]. Although most of these mobile devices use Multiprocessor System-on-Chip (MPSoC) platforms, different kinds of applications such as gaming, navigation, social media applications, and etc. are running on them; consequently, they are used like a general purpose processor. There is also an increasing demand for data centers and cloud resources which are servicing different types of customers and applications simultaneously. It's expected that many-core platforms, forming single-chip cloud computers such as [68], increase the computation capabilities. As an example, Figure 1.2 illustrates Intel's SCC processor organization, which shows a 24 processor laid out as a 6×4 tiled grid.

The prevalence of mobile devices and cloud computing brings a unique execution model to many-core systems where different applications can be executed concurrently on many-core platforms. These applications can be downloaded or uploaded from any source and usually no offline analysis of them is available. They can enter and exit at any time. In this execution model, resource assignments cannot be done at design time and the resource assignment decisions should be done at runtime. Without careful design of the runtime system, the efficient use of resources cannot be achieved in many-core systems.

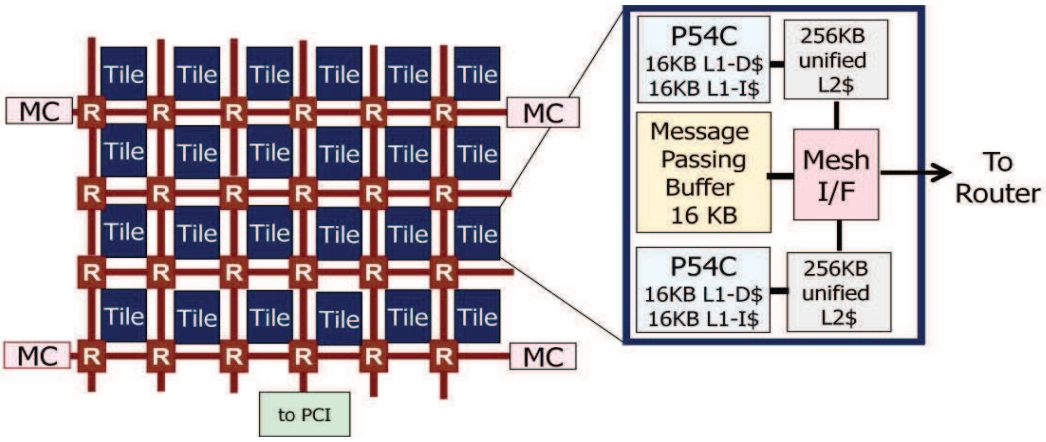


Figure 1.2: SCC Processor layout – 24 Tiles in a 6x4 mesh of routers (R), four DDR3 Memory controllers (MC) and a single PCI link. Each tile contains 2 P54C cores, 16 kB each of data cache (L1-D$) and instruction cache (L1-I$) plus a 256 kB unified cache. The tile includes a 16 kB Message Passing Buffer and a mesh interface unit (I/F) to connect to the router [126].

Efficient use of hundreds of cores on a chip comes with unique challenges. Not addressing these challenges might yield a very poor performance in many-core platforms. Using traditional bus-based platforms is not practical due to high power consumption and latency, and bandwidth limits which make those platforms unscalable. A proper interconnection network has to be used for connecting different cores. Developing efficient programs to leverage many-core resources is not straightforward which makes programmability a great challenge for many-core platforms. The necessity of sharing memory between different cores, makes the memory management of many-core systems very complicated.

1.2 Memory Subsystem Challenges in Many-core platforms

The concept of the *Memory Wall* has been known for decades [130] and it states that the rate of improvement in processor speed is higher than that of off-chip memory (e.g., DRAM) speed, therefore the memory system is the performance [and power] bottleneck for the entire

system. To tackle this problem, a hierarchy of memories with different size/speed is created. Registers, multiple levels of onchip memories (cache or SPM), RAM, and storage(disk) are four different levels of memory which can be found in most computer systems. A memory management system decides how to use the memory subsystem – from small and fast registers to large and slow disks. Traditionally caches are used to fill the speed gap between memory and logic. Fast SRAM based cache memories are controlled by hardware and they are transparent from software. In presence of more than one core, it's very common to have both private and shared caches. Different levels of cache create a tradeoff between capacity and speed. Figure 1.3 depicts a typical memory subsystems in multicore platforms. While all requirements of traditional memory subsystems such as high bandwidth, low energy, and low cost exist in many-core system, some requirements and challenges arise while moving toward hundreds of cores.

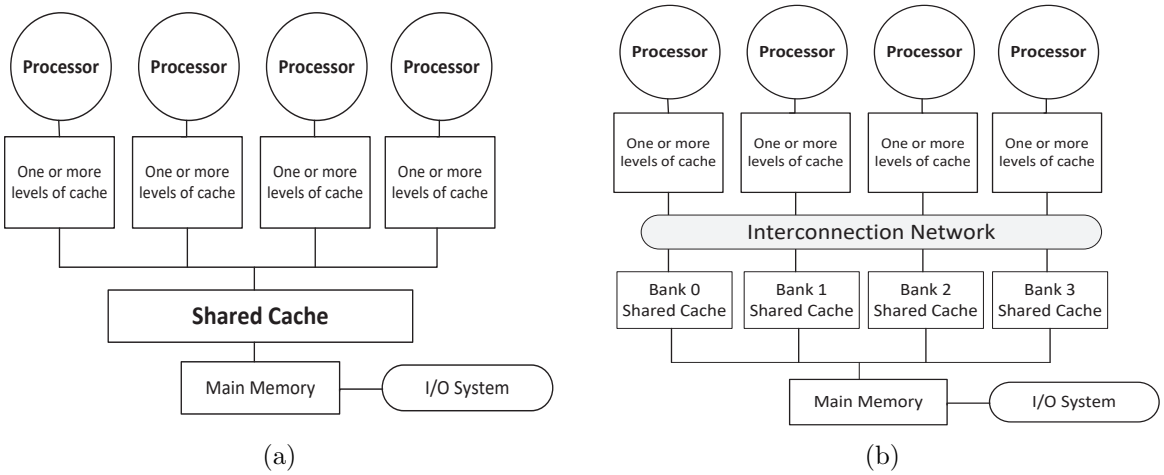


Figure 1.3: Basic structures of single-chip shared-memory multi-cores using (a)bus (b) interconnection network [61].

Although processor frequency hasn't improved for many years, multi and many core systems create the necessity for higher Memory bandwidth while DRAM doesn't scale in nanometer feature sizes [94]. Latency and bandwidth of accessing offchip memory depends on the distance to memory controller and number of cores per memory controller. To compensate this gap between processor and memory performance, huge real estate of onchip

transistors are dedicated to memories and memory is the major shared resource among different cores [93]. Concurrently running applications don't have the same memory requirements. All these characteristics make the management of memory subsystem in many-core platforms very complicated.

In many-core systems, coherency is a major bottleneck for scalability. If a piece of data is copied in different cores, the consistency of those copies should be guaranteed by the coherence protocol. If any thread updates one of these copies, all other copies should be invalidated which is very costly [135]. It's shown by [21] that even a single contended cache line can destroy the potential scalability of an application (Figure 1.4).

As we scale to many-core systems, it becomes increasingly challenging to scale the corresponding cache-based memory hierarchies [58, 1, 85] . One important reason is the rapid increase of coherence logic overhead with the number of cores. Some processors have already tried to alleviate this problem by removing hardware cache coherence from processors either partially or completely, e.g., Intel SCC [69], Kalray MPPA-256 [36]. In these architectures, the coherence whenever needed by the application/system must be implemented in software. Another promising solutions to overcome this problem is to use *Scratchpad Memories (SPM)*, also known as Software Controlled Memories and Software Programmable memories, instead of caches [111]. SPMs are controlled by software and in contrast with hardware controlled caches do not require a hardware coherence protocol, but instead transfer the burden of memory management to software. Different platforms such as IBM cell processor [66] and Tilera [124] use software controlled memories as their onchip memory.

SPMs offer many advantages over caches:

- The use of SPMs allows developers to exploit application semantics effectively to achieve efficient execution.
- They provide power efficiency by eliminating the hardware overhead of traditional

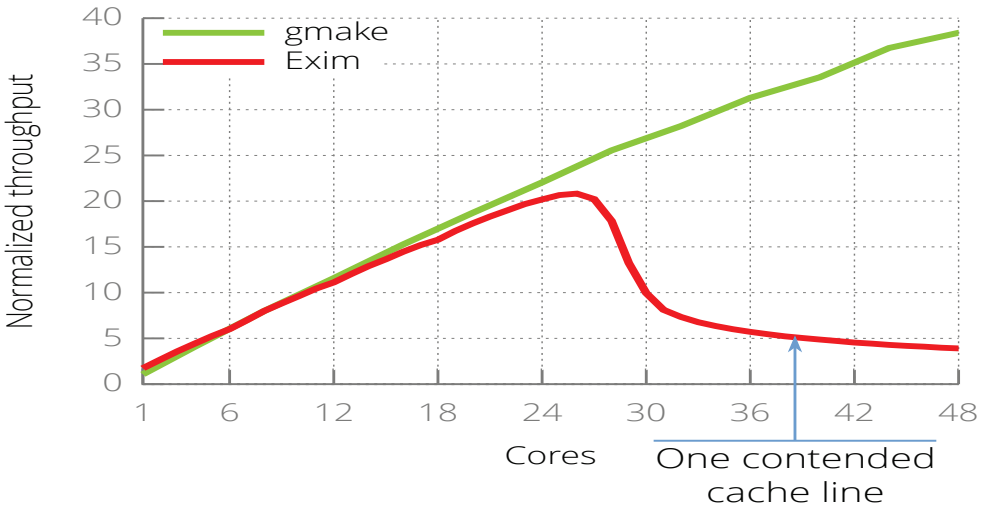


Figure 1.4: Effects of cache contention on scalability [21].

caching.

- Predictability is a critical factor for real-time systems. It is hard to estimate the worst-case execution time (WCET) of software executing in cached architectures, since cache replacement policies executing in hardware result in unpredictable execution times for cache hits and misses. On the other hand, SPMs allow predictable estimation of WCET since all memory accesses are explicitly controlled in software.

- There is potential for performance improvement by orchestrating the management of data transfers explicitly in software.

- Present the ability to explicitly manage data accesses for thermal and wearout constraints, particularly for emerging memory technologies, e.g., non-volatile memories (NVM).

While there is a large body of work on using SPMs for guaranteeing WCET (e.g., for real-time applications), this thesis focuses on exploiting SPMs as an alternative selection of onchip memories in many-core systems to avoid coherency problem.

Early efforts in programming SPM-based architectures required application developers to

insert data management instructions manually. However, with the increasing complexity of embedded software [41], as well as the diversity of the underlying architectures, automated techniques are required to understand the application and insert data management instructions automatically. Automatic insertion of data management instructions can be achieved statically (by programmers or the compiler), or dynamically (through runtime systems that execute additional instructions to achieve the desired effect).

SPM virtualization can be used to design effective memory hierarchies and provide applications with the convenience of a transparently managed address space, while simultaneously using developers' guidance to mitigate the complexity of managing shared memory, as well as utilizing the non-uniform characteristics of the underlying hardware. These techniques are designed to wisely allocate SPM space among tasks in the multitasking environment. When software-programmable on-chip memory is contained in an additional layer of virtualization, application developers can specify when and what data to store near the core executing its instructions without knowing the intimate details of the underlying memory architecture. The runtime software (as part of the operating system) can map the data to any physical location on- or off-chip. Additional address translations to this intermediate virtualization layer must be stored to enable accesses to the data in the address space. This can be done with a translation table that maps virtual addresses of the task executing on the core to intermediate physical addresses, which is essentially a physical address in on-chip SPM space.

Bathen et al. [11] introduced the concept of SPM Virtualization through SPMVisor, a scheme for bus-based multi-cores in which virtual SPMs (vSPMs) allow programmers to assume access to the entire on-chip memory space through virtualized address spaces. Each thread can therefore assume it has access to a dedicated contiguous memory (Figure 1.5b) without considering interference from competing threads (Figure 1.5a). A virtualization layer, with the help of of Protected Evict Memory (PEM) space, is responsible for determin-

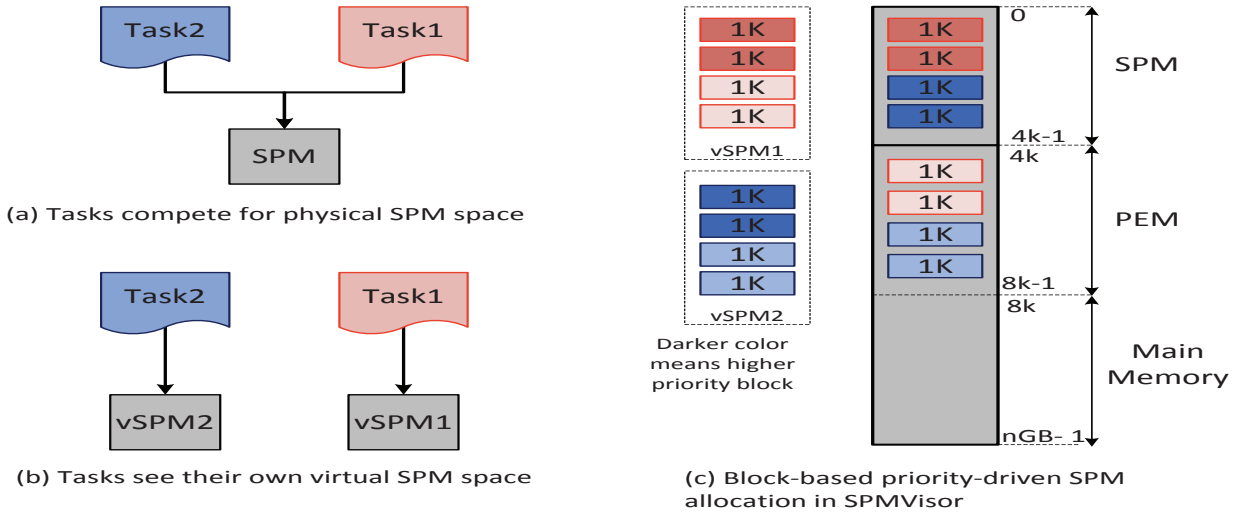


Figure 1.5: Concept of virtual SPMs in SPMvisor [11].

ing what data is placed on-chip and what data is placed off-chip based on defined priorities (Figure 1.5c). This bus-based approach does not work in platforms with tens or hundreds of cores where a bus should be replaced with an interconnection network: using PEMs becomes very costly, and obtaining priority of applications is not always possible. But this SPM virtualization can be adapted to many-core platform characteristics.

When platform size approaches hundreds of cores, a single central manager becomes incapable of managing the entire system with a reasonable overhead due to increased communication distance, network congestion, overhead of keeping data for every application, computation, etc. Therefore, using central management for memory mapping degrades system performance significantly and makes any management system unscalable. To solve this problem, a distributed management should be employed to increase the efficiency of resource assignment, especially for SPM resources in many-core systems.

1.3 Variability

Advances in the semiconductor industry have allowed creation of smaller feature sizes for transistors. This process has shrunk to the atomic scales. For example, in a 22nm manufacturing process, the gate length is only 42 atomic diameters and the oxide is only five atomic layers thick [54]; and this gets even worse in newer technologies. Atomic feature sizes makes the fabrication and control process very hard and costly for hardware manufacture companies. As a result, there is not a total control on fabrication process and some intended characteristics differ from the final product, resulting in various forms of manufacturing or process variability.

Within die and die-to-die process variations are produced by systematic effects (such as lithographic lens aberrations) and random effects (such as dopant density fluctuation) [104, 95]. One of the most important parameters, prone to variation, is transistor threshold voltage (V_{th}). Variation in threshold voltage affects frequency and leakage power of processors [104]. Voltage and temperature are two other important factors contributing to variation.

Speed (frequency) variation between different parts of the die is one of the most important effects of process variation. Lower voltage, higher temperature, and slower transistors can make some parts of the die slower than others. Significant frequency difference already exists between different cores in multi and many-core platforms. For example, Figure 1.6 shows the frequency variation for the Intel 80-core TeraFLOPS processor, measured for different operating voltages [39]. As demonstrated, there is a 28% frequency variation between fastest and slowest core at 1.2V, this number is 62% at 0.8V.

In addition to PVT – Process, Voltage, Temperature – variation, considerable variation exists in the software stack. Two different applications may have completely different processing, memory, and IO requirements. Even a single application can experience variation in data, control flow, technology, and environment [6]. Not addressing the variation in software

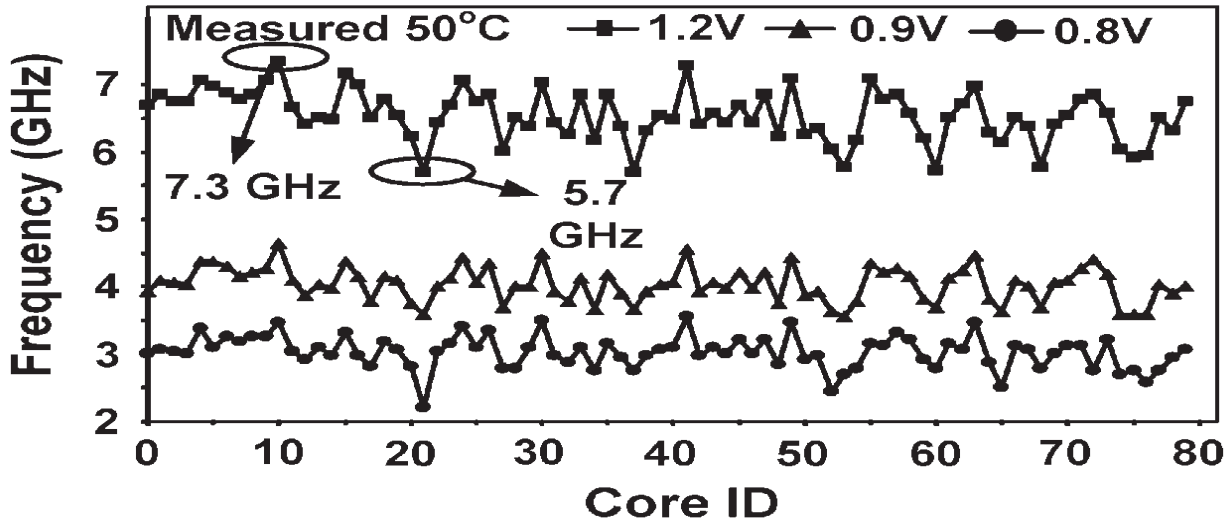


Figure 1.6: Measured core-to-core max frequency variation for 80 cores on a single die [39].

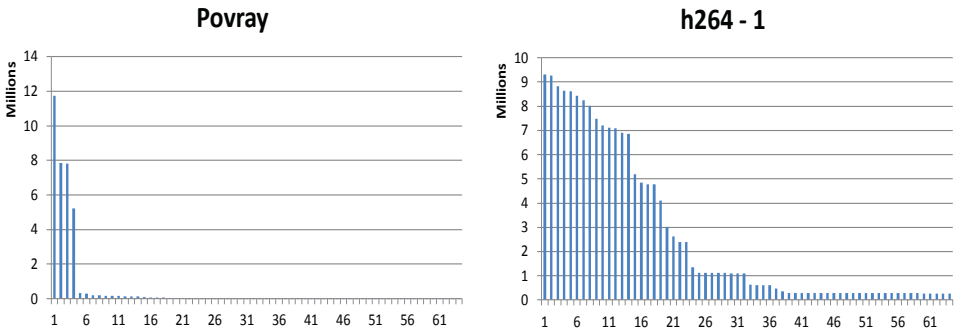


Figure 1.7: Highest accessed pages of povray and h264 benchmarks

stack can cause a significant performance degradation. Memory access variation is of great importance specifically, because memory is usually the main bottleneck of performance.

For example, Figure 1.7 shows the number of accesses to the highest accessed pages for the h264 and povray benchmarks. Povray has a few pages with very high number of accesses and some pages with very low number of accesses. While povray has a discrete spectrum of accesses, h264 has a relatively continuous spectrum and the number of accesses to each page can be high, medium, or low. Also Figure 1.7 suggests that h264 requires more memory resources than povray, indicating significant variation in memory demand across applications.

Any application can experience variant behavior and characteristics during its course of

execution. Therefore the average or aggregate behavior of an application is not a good representative of the application. For instance, if page A is accessed more than page B during the execution of an application, it's quite possible that in a finer grained analysis of memory accesses, a long period of time exists in which page B is accessed much more than page A. Resource allocation and optimization can be more efficient considering the temporal variation in a single application. Applications have very dynamic behavior and the memory requirements of each individual application varies over time throughout the course of execution. For instance, rendering high-motion scenes needs more memory resources than rendering still scenes. Most applications can be divided into different phases of execution. Repetitive, or *phasic*, behavior of applications has been the subject of research for more than a decade. Typically the phasic behavior is investigated with respect to the structure of applications (i.e., application's basic blocks and control flow) or execution working set. Although these efforts have proven useful, detecting patterns in the execution of an application by tracking instructions (or Basic Blocks) does not always capture the memory phases of the application. Typical program phasic behavior extraction techniques do not differentiate between program and memory behavior [9, 40, 110], and thus miss opportunities for more aggressive memory management in the face of high memory demands.

1.4 Thesis Overview

Any architecture and management system in many-core platforms should address challenges in the memory subsystem. Most of these challenges are specific to many-core platforms and management systems for single and multi-core platforms do not provide a promising solutions for them. Some examples are listed below:

- Memory Coherency: Sharing memory is a crucial technique to speed up communication between different threads or processes, specially in multi-threaded applications.

Caching of shared data creates the need for implementing coherence protocols which is a huge burden for the system and can cause a significant performance degradation. Using traditional caching systems has the potential to make the many-core platform inefficient.

- Dynamic Workload and lack of static analysis: With advances in technology and mobile devices, powerful computing systems are ubiquitous now and they are exhibiting more cores every year. These devices have a very dynamic workload and any combination of applications can be executed on them. While static analysis is available for some applications, there is no prior applications available for most applications. Therefore many management decisions should be made at runtime.

- Memory Requirement Variation: Different Applications concurrently running on a many-core platform have different memory requirements. This memory requirement variation also exists within each individual application. Assigning predefined memory resources to each application reduces the efficacy of resource management. Sharing physical onchip memory resources and providing the ability to access remote cores for each thread, alongside with detecting and exploiting the workload variation increases the performance of many-core platforms.

- Scalable Management: With increase in the number of cores, employment of a central manager becomes excessively expensive. Increased communication distance between cores/applications and manager, network congestion due to excessive number of messages from and toward the manager, increased required storage to keep data for every application, rise in amount of computations, etc. contribute to the intolerable overhead of central management in many-core platforms. Therefore, a distributed management is necessary for scalable memory management in many-core platforms.

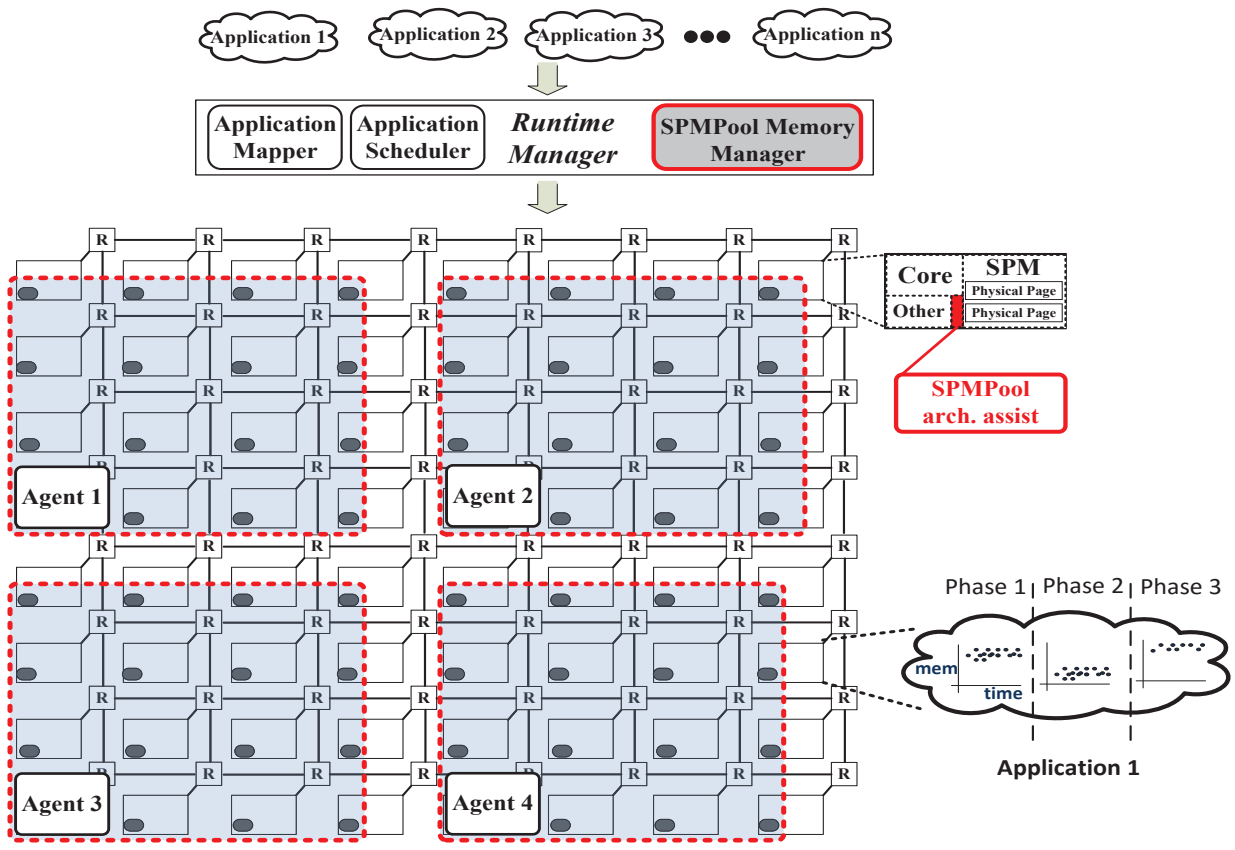


Figure 1.8: System level view of SPMPool

A comprehensive solution for these challenges needs a powerful runtime system equipped with architectural assists. To address all these challenges this thesis proposes SPMPool: a scalable platform for sharing scratchpad memories. Figure 1.8 depicts the system level view of SPMPool. Scratchpad memories (SPMs) are used in each core as the onchip memory – instead of caches. SPMPool approach exploits the underutilized memory resource by dynamically sharing SPM resources across cores, and adapts to the overall memory needs of multiple applications that are concurrently executing on the many-core platform. SPMPool is the first solution for dynamic runtime memory management by sharing SPMs across concurrently executing applications in unpredictable workloads.

The SPMPool Memory Manager organizes SPM resources as Pools-of-SPMs and assigns the SPM resources to applications executing within the pool based on their memory require-

ments. SPMPool Architectural Assists facilitate the sharing and remote access of SPMs by applications. Each core executes a memory phase change detection scheme and sends the information to the associated manager. The objective of SPMPool approach is to maximize system performance by minimizing the wait time incurred by memory accesses. The memory mapping is managed by a distributed management scheme which makes this approach scalable.

Different aspects of SPMPool approach is described in the following chapters of this thesis. We start with some simplifying assumptions to illustrate the basic concepts of SPMPool. After showing the effectiveness of SPMPool, some assumptions are relaxed to address the shortcomings of the basic SPMPool approach. The organization of the rest of this thesis can be summarized as follows:

- In Chapter 2 the basic concepts of SPMPool approach is described. SPM resources are shared between cores, creating one core with a central manager. The required architectural assists and memory manager routines are characterized. Memory mapping problem and some optimal and heuristic solutions are discussed. A solution for sharing memory without implementing coherence protocol is introduced in this chapter.

- In Chapter 3 an auction-based distributed memory management scheme is proposed. After proving the efficacy of auction-based method in solving memory mapping problem of SPMPool, a distributed management for hundreds of cores with numerous pools-of-SPMs is introduced.

- In Chapter 4 offline and online methods to detect memory phases are introduced. Those memory phases are used to improve the memory mapping in SPMPool by making the mapping more dynamic. Prioritizing different applications which is a byproduct of memory phase detection is used to eliminate offline analysis of applications.

- In Chapter 5, the thesis is ended with concluding notes and future directions.

Chapter 2

SPMPool

Runtime SPM Management in Embedded Many-Cores

2.1 Introduction

As mentioned in Chapter 1, embedded many-core systems increase their performance by offering highly parallel computational resources. This allows embedded systems to run demanding applications such as real-time video processing, live object tracking, etc. With increase in the number of cores on chips, keeping the coherency of caches becomes very challenging. In addition, the power consumption of caching logic hardware makes the cache architecture very power hungry in many-core platforms. As an alternative, Software Programmable Memories (also called scratchpad memories or SPMs) are deployed as on-chip memories in embedded many-cores for achieving lower power, better predictability, and higher area efficiency. Such a large number of SPMs distributed over many cores makes memory mapping challenging for the controlling software.

The high variability of memory intensity between concurrently executing applications causes on-chip memory resources to be more valuable for some applications over others.

Furthermore, the abundant presence of processing elements in many-cores results in periods of execution during which not all cores (as well as their SPMs) are utilized. However, traditionally applications are limited to using only their local memory [26, 7, 8, 30]. On the other hand, dynamic scenarios – where applications start and stop at any time – can result in changing memory access patterns that require data allocation to be adapted at runtime.

In this chapter, we present the SPMPool approach [121] that exploits the underutilized memory resource opportunity by dynamically sharing SPM resources across cores, and adapting to the overall memory needs of multiple applications[1] that are concurrently executing on the many-core platform. To the best of our knowledge, SPMPool is the first solution for dynamic runtime memory management by sharing SPMs across concurrently executing applications in unpredictable workloads[2].

Figure 2.1 depicts the system level view of SPMPool. Applications start and stop at any time and are assigned to cores within a pool by the *Runtime Manager*. Multiple applications can be assigned to a pool. SPM resources in a pool are shared among all the applications assigned to that pool. The *SPMPool Memory Manager* organizes SPM resources as *Pools-of-SPMs* and assigns the SPM resources to applications executing within the pool based on their memory requirements.

SPMPool Architectural Assists in Figure 2.1 facilitate the sharing and remote access of SPMs by applications. Whereas multiple design objectives can be addressed, in this work the objective of SPMPool approach is to maximize system performance by minimizing the wait time incurred by memory accesses. This is done by assigning underutilized SPM resources of a pool to memory-intensive applications within the pool.

The rest of this chapter is organized as follows: In Section 2.2 the motivation for sharing SPM resources is discussed. Section 2.3 summarizes different SPM related research efforts in

[1] we use the terms application and task interchangeably in this work.

[2] The set of running applications and their entrance times are not known prior to execution.

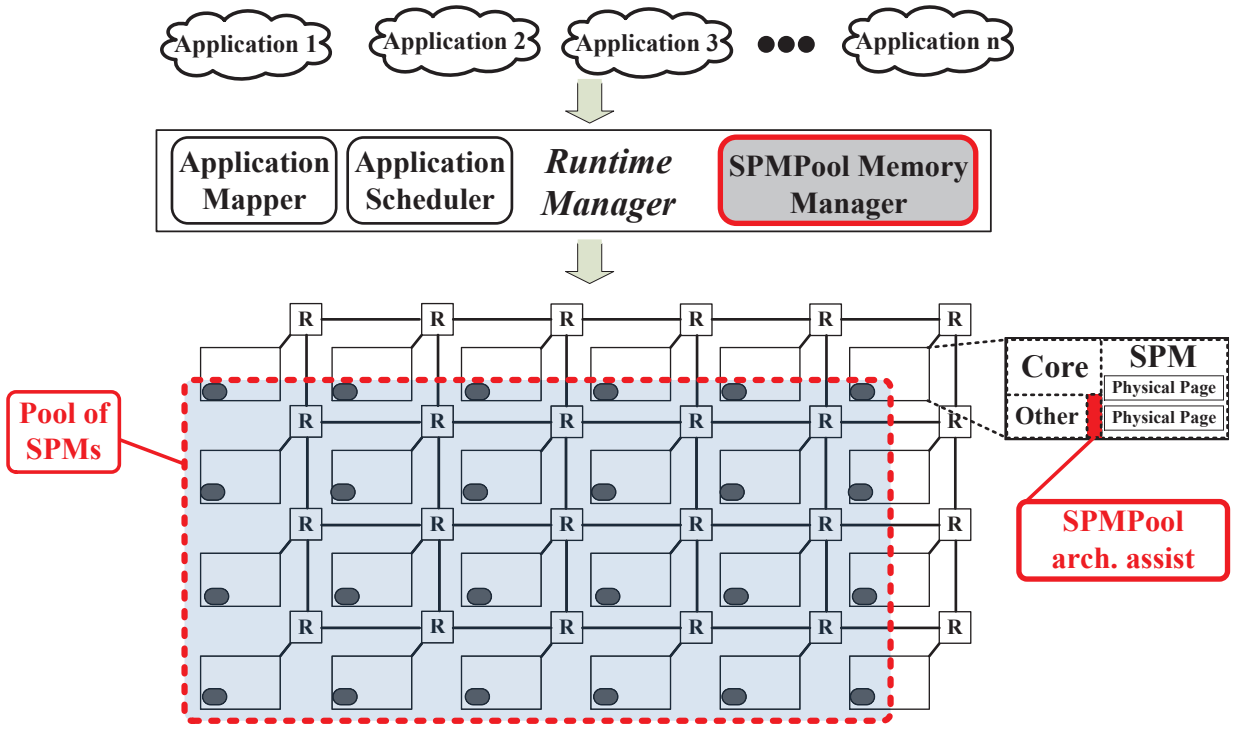


Figure 2.1: SPMPool role in many-core system with a single-pool configuration.

recent years. Section 2.4 introduces basic architecture and different components of SPMPool. Runtime system requirements, architectural assists, and memory mapping problem are also discussed in this section. Section 2.5 presents experimental setup and results, while Section 2.6 will discuss the scalability and task mapping in SPMPool. Section 2.7 contains some concluding remarks.

2.2 Motivation

In memory intensive applications, the amount of data used exceeds the amount of SPM resources on-chip – many applications use megabytes of data while SPM resources are on the order of kilobytes. In order to avoid severe performance degradation, the subset of data stored in SPM must be carefully selected. The state-of-the-art mechanisms for allocating data to SPM select this subset based on the size of the SPM available at the core on which

the respective application is executing [98, 114, 127]. However, if there are fewer applications than cores, the SPM resources belonging to unused cores are unutilized. Another form of underutilization of SPM resources occurs when concurrently executing applications have considerably different levels of memory usage.

For instance, users can run various types of applications concurrently on a mobile device. Mobile computing devices like tablets are replacing PCs and they have very powerful processors. In these devices many processes are running simultaneously. Its very likely that in the future these mobile computing devices use many-core platforms. In these platforms, it is possible to browse the web, listen to music, find a place using the map, play a game, video chat with a friend, or use a calculator to compute a mathematical expression. These applications have different memory requirements; some of them are very memory intensive (e.g., imaging applications) and some of them have small memory footprint (e.g., calculator).

In both cases of underutilization, severe performance penalties can result from inefficient SPM usage. A solution to this problem could be the redistribution of SPM resources to executing applications with the goal of reducing the overall memory latency. In this work, we model memory access latency for one application as the summation of access latency over all its memory accesses during the course of execution. Overall memory access latency can be obtained by adding the memory access latency of all applications in the working set (we use the overall memory access latency as a proxy for performance). To exemplify, consider a multi-core architecture with a 4-core mesh network as illustrated in Figure 2.2.

Now consider a sample workload (shown in Figure 2.3) on a smart-phone consisting of three applications with a working set of four virtual pages each. Each application has a different memory footprint. Application B is an imaging application with high memory usage, while application C is a scientific calculator and has low memory requirements. Application A is a text editor and has moderate usage of memory. In Figure 2.3, each page is annotated with the total number of accesses. All three applications start executing concurrently. Figure 2.4a

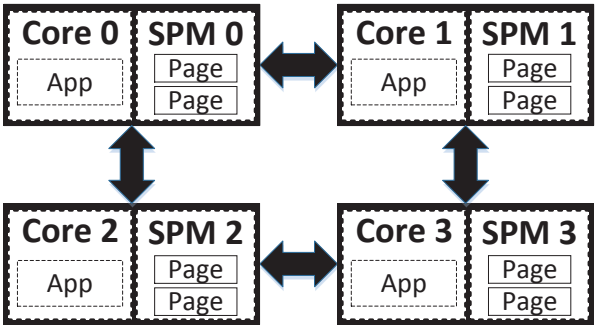


Figure 2.2: Example of a system with four cores and four SPMs.

Application X	App A	App B	App C
Page 0 # accesses	Page A0 75	Page B0 1000	Page C0 5
Page 1 # accesses	Page A1 100	Page B1 900	Page C1 5
Page 2 # accesses	Page A2 50	Page B2 900	Page C2 5
Page 3 # accesses	Page A3 5	Page B3 1000	Page C3 10

Figure 2.3: Applications and their working set of pages, along with the total number of accesses to each page over the entire execution.

shows the resulting memory mapping for this working set on the system, if each application was to use its local SPM only.

A superficial inspection of this memory mapping reveals that application B is forced to place two highly utilized memory pages in off-chip memory, while physical SPM pages belonging to Core 3 are unused. Also, application C's relatively low utilized pages are granted an equal amount of SPM pages as other applications' more utilized pages. Memory access latency for each application is obtained by:

$$\sum_{p \in application\_pages} (\#accesses\_to\_p) \times (Latency\_of\_access\_to\_p)$$

The latency of access to page p depends on the distance between an application's home core and page p. The total memory access latency (based on the information in Figure 2.3 and Table 2.1) for application A is 5125 cycles, application B is 164000 cycles, and application C is 915, resulting in a total memory access latency of 170040 cycles after all applications

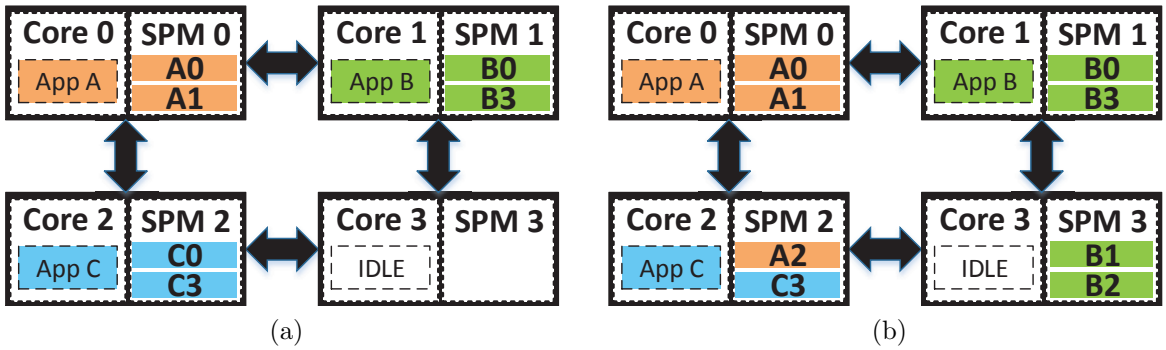


Figure 2.4: Motivating example for sharing SPM resources.

Table 2.1: Cost of Access (Number of Cycles) to each memory

	SPM 0	SPM 1	SPM 2	SPM 3	Off-chip
Core 0	1	4	4	7	90
Core 1	4	1	7	4	90
Core 2	4	7	1	4	90
Core 3	7	4	4	1	90

complete their execution.

If we expanded the pool of available memory for each application by allowing them to use non-local SPMs, we could place memory pages on-chip according to their utilization and use all SPM resources. Using this approach, the alternative configuration shown in Figure 2.4b successfully reduces the total amount of off-chip memory accesses for all applications to 11390 cycles. We propose to combine all SPMs in this manner as a global Pool-of-SPMs that provides memory resources accessible to any executing application.

Current allocation methods either completely ignore any potential contention for memory resources from other applications, or assume the set of executing applications at any given time is predictable [125, 116, 122, 50, 97, 90, 133]. When an application is started or stopped, previously unused data may be accessed frequently and data that has previously been allocated to SPM may no longer be accessed. Hence, the state-of-the-art techniques can hardly avoid severe performance penalties in such scenarios. SPMPool addresses such issues that arise when applications start or stop at any time and may execute concurrently.

2.2.1 Key Contributions of SPMPool

- We present a *Runtime Memory Manager* that evaluates and generates performance-driven SPM mappings for concurrently executing applications on application start/stop.
- We describe exact and heuristic formulations for the SPMPool Mapping Problem to efficiently share SPM resources among applications.
- We illustrate the architectural assists needed to enable pooling of SPMs for local and remote SPM accesses.
- We demonstrate efficacy of SPMPool for a mix of concurrently executing applications on configurations ranging from 16 to 256 cores, showing up to 76% reduction in memory access latency with minimal overhead.

2.3 Related Work

A variety of approaches for both allocating and scheduling spatially shared resources in multi-core processors have previously been explored [107, 3, 2, 63, 136, 78, 10, 91]. Some of these approaches propose scheduling algorithms and migration policies for workloads on multi-cores that use power and performance counters to determine task to core mappings (e.g., [136, 78, 10, 91]). These approaches all target traditional power or performance metrics, monitor shared last level cache (LLC) contention, and improve individual application behavior more significantly than the overall workload. Thread-scheduling can be used orthogonally to resource partitioning in order to alleviate contention for certain shared resources such as interconnects, but does not directly allocate contended resources such as memory. Hoffman et al. [63] present a novel programming model that, among many features, provides a decision-making engine in order to adapt applications' hardware allocation to meet goals specified by each application developer. More recent work ([107, 3, 2]) has focused on multi-

core architectures that include a shared reconfigurable fabric. They specify runtime managers that adaptively allocate subsets of the reconfigurable fabric between concurrently executing tasks in variable workloads. While we can draw inspiration from some of these techniques and employ others in conjunction, the general resource sharing techniques for multi-cores are not explicitly applicable for sharing SPMs. Most of the approaches depend exclusively on performance counters, including LLC miss rate, to evaluate their allocations. These counters are architecture-dependent, and depend specifically on a cache memory hierarchy that is not analogous to an SPM-based architecture.

As for allocation of shared caches for contentious workloads, there has been significant interest in performance-aware cache partitioning [113, 99, 24, 74, 100, 72, 131]. The cache community has previously explored distance-aware dynamic partitioning (NUCA) [76, 29, 57] as well as sharing cache resources for multi- and many-core architectures [108, 84, 83]. These approaches, similarly to SPMPool, typically aim to reduce the number of off-chip memory accesses. However, these solutions are not applicable for sharing SPMs: cache contents are controlled by hardware implemented replacement policies and address mapping, while SPM contents are explicitly allocated and mapped freely by software. SPM organization is not hierarchical and does not hold duplicate data, and is therefore more similar to a swap-space. Moreover, cache hierarchies are unable to address the scalability issue for chips with hundreds of cores executing numerous applications all sharing the entire on-chip memory pool. More recently, Beckman et al. [12] have proposed a solution for software-controlled partitioning of shared last level caches in multi-cores.

Along with LLC, DRAM is one of the most commonly shared resources between applications in many-cores. There exist parallels between the problems outlined in this work and those that the NUMA community have long been addressing for main memory. They have previously proposed OS-controlled mapping of data in memory physically near the accessing tasks in order to both increase performance [20] and reduce power [82]. They also attempt

to characterize the complex memory access costs for modern NUMA architectures [22]. We share a common goal with NUMA solutions of performance improvement through memory access latency reduction. A network of on-chip SPMs can be likened to a NUMA architecture with numerous latency levels. In the case of SPMs, migrations can be made at a finer granularity (e.g., cache-line vs. page), and therefore may be more sensitive to access locality.

There is a large body of existing work on SPMs that includes uni- and multi-core solutions for SPM mapping, covering static or dynamic approaches. Static SPM mapping algorithms map a fixed subset of the memory address space to SPM space at compile-time, and the contents of the SPM do not change during execution [127, 114, 115, 98]. Such rigid techniques typically target guaranteed worst-case execution times (WCET) for real-time applications and are vulnerable to the changing needs of unpredictable workloads consisting of multiple applications that start and stop unpredictably.

Dynamic SPM mapping techniques can be further divided into compile-time-driven dynamic approaches, and run-time approaches. Compile-time-driven dynamic SPM algorithms typically consider only a single application, or assume predictable workloads for multiple applications, which is unrealistic [125, 116, 122, 50, 97, 90, 133]. More recent run-time approaches augment dynamic compile-time mapping with a combination of hardware and operating system (OS) support to allow for reaction to runtime memory behavior of individual applications [48, 28, 11, 43, 37, 75]. However – unlike our SPMPool approach – these solutions do not consider simultaneous (spatial and temporal) sharing of the SPM space with other executing applications at runtime or do not provide solutions for multicore platforms.

To the best of our knowledge, for many-core systems with unpredictable concurrently executing workloads, SPMPool is the first SPM mapping technique that shares distributed on-chip memory resources across multiple executing applications. Furthermore, SPMPool adapts the mapping at run-time based on the relative memory needs of the concurrently-executing applications. In this work, memory requirements of applications are obtained by

offline profiling; but this can be replaced by any technique that quantifies memory requirement of applications. Our SPMPool approach complements other adaptive resource sharing/management approaches targeting compute-focused paradigms such as invasive computing [60, 107], by providing an adaptive approach for dynamic memory management/sharing.

2.4 SPMPool

Recall from Figure 2.1 that we apply the SPMPool approach to a many-core platform consisting of tiles connected through a Network-on-Chip (NoC), each containing a simple core and a local SRAM scratchpad bank. This platform supports workloads consisting of many applications with different memory characteristics, and includes a runtime manager to handle application scheduling.

Many modern mobile platforms support the concurrent execution of multiple independent applications on both shared and separate resources. In many-core systems, coherency is a major bottleneck for scalability. If a data is copied in different cores, the consistency of those copies should be guaranteed by the coherence protocol. If any thread updates one of these copies, all other copies should be invalidated which is very costly [135]. It has been shown by Clement et al. [31] that even one single contended cache line can destroy the potential scalability of an application.

To avoid this complexity, in our proposed architecture only a single copy of a piece of data may exist on-chip. We enforce this in the following ways:

1. Single-threaded applications run on one core.

2. Multi-threaded applications in which different threads do not share memory can be modeled and executed as multiple single-threaded applications on different cores.

3. For multi-threaded applications in which different threads share memory, only one copy of shared data will be exist in on-chip memory at any given time.

This architecture might degrade performance in some multi-threaded applications, but it obviates the need for a coherence protocol.

In this initial investigation we assume a flat memory hierarchy, i.e., data is not cached. Therefore, only one copy of any piece of data is held in on-chip memory, eliminating the necessity for us to implement any additional coherence protocol. This simplification allows us to explore diverse workloads at different scales in our experiments. The previous assumptions outline the framework in which we apply the SPMPool approach for the remainder of this work.

As shown in Figure 2.1, SPMPool consists of three different components:

1. *Pools-of-SPMs* in the tiled many-core computing platform with a NoC fabric, where applications mapped to each pool can share SPMs based on their varying workloads and dynamic memory needs. Each application has a private virtual address space, not shared with any other application.

2. *SPMPool Memory Manager*, which determines the placement of each executing application's memory working set within a Pool-of-SPMs, while considering the memory requirements of all other applications executing within that Pool.

3. *SPMPool Architectural Assists* within each core, to enable sharing and accesses to remote SPMs within the shared Pool-of-SPMs.

We now describe the *SPMPool Memory Manager* (outlining exact and heuristic memory mapping algorithms), and the *SPMPool Architectural Assists* (for supporting intra-Pool sharing and remote SPM access).

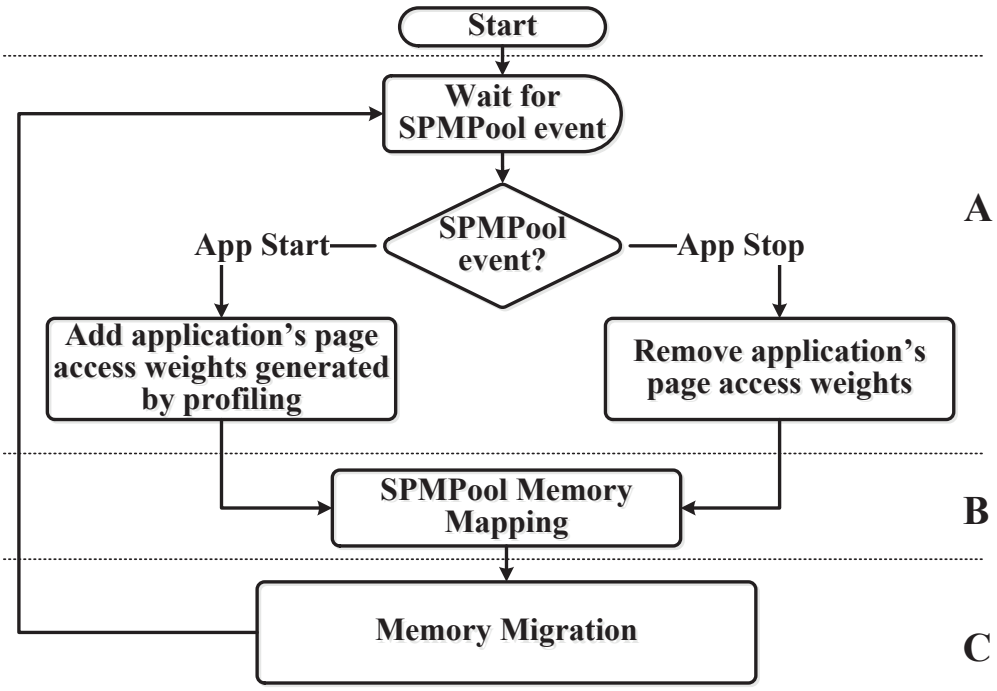


Figure 2.5: SPMPool Memory Manager

2.4.1 SPMPool Memory Manager

Figure 2.5 gives an overview of the SPMPool Memory Manager. We assume that each application's working set (set of pages accessed throughout execution) is tagged with a *memory access weight* representing the number of accesses to each page (generated through a preprocessing step).

In Phase A of Figure 2.5, the SPMPool Memory Manager is triggered by an SPMPool Event that corresponds to the application start or stop within the runtime system. For an application starting or stopping execution, SPMPool adjusts the internal bookkeeping to update the memory access weights corresponding to the working set.

In Phase B of Figure 2.5, the SPMPool Memory Manager generates a new memory mapping to share the SPMPool resources across the active applications within the Pool-of-SPMs. The output of Phase B is a virtual-to-physical memory map for each application with the

goal of reducing overall memory access latency of all applications within the Pool-of-SPMs.

Finally, in Phase C of Figure 2.5, the SPMPool Memory Manager performs memory migration for applications whose logical-to-physical address mappings have been modified. The runtime manager sends the changes in memory mapping to the affected applications as updates for their local translation tables (more details in Section 2.4.2). The Memory Manager triggers the physical memory movement, which includes the write-back to main memory of evicted SPM pages.

In the remainder of this section, we will explain each aspect of the SPMPool Memory Manager in more detail.

SPMPool Event

An SPMPool event is defined as a change in the workload that triggers the runtime manager to update the memory mapping. As previously mentioned, the event model consists of two types: application start and application stop. When either event occurs, there are some SPMPool-specific tasks to complete before the new memory map can be generated. In the case of application start, the runtime manager initially assigns application threads to random tiles with available compute resources. The manager also receives an incoming application's page list and stores all lists for executing applications. This SPMPool-specific information is generated at compile-time, and is discussed later in this section. In the case of application stop, the application's virtual pages are de-allocated from physical SPM pages and written back to main memory.

SPMPool needs to perform some preprocessing on application binaries to determine the working set of memory pages and the access pattern to those pages for each application. For every page in an application's working set, an ordering of accesses to the given page is created. This is done by scanning a memory trace of a sample execution for the given

application. In order to assign a weight to data memory in advance, some information about that memory must be accessible. In this work, we assume that by profiling the applications offline, we can extract access information for data memory. The shortcoming of this approach is that by changing the inputs, control flow of the program or the address size and range of dynamic memory allocated with malloc will be changed. This problem is the primary focus of our current work in progress.

Using the page access information, a weight is generated for the page - in this chapter, this weight is simply the total number of times the associated page is accessed over the application's entire execution. In multi-threaded applications, each thread has an individual weight list. If a page is shared between different threads of an application, its weight will be the aggregation of accesses from all threads. This page-weight will be included in the list of the thread with the most number of accesses to that page and tagged as a shared page. In the other words, we have only one entry for each shared page in the application page-weight list. The application's list of pages and associated weights is then passed to the SPMPool runtime upon the application's entry into the system.

SPMPool Memory Mapping

Here we focus on the SPMPool Memory Mapping step (Phase B of Figure 2.5) outlining exact and heuristic formulations. The goal of SPMPool Memory Mapping is to select a subset from the many pages of concurrently executing applications and assign them to the limited number of on-chip SPM slots in order to minimize overall memory access latency (summation of all memory access latencies over every application in the working set during their course of execution). Remaining pages stay in off-chip memory. To make this selection feasible it is necessary to characterize the memory behavior of each application for comparison. For this work, we simplify this characterization by assigning each page its memory access weight that represents the number of times that page is accessed.

The exact formulation of the SPMPool Mapping Problem can be expressed as outlined in Optimization Problem 1. Note that for the completeness of memory mapping, main memory is also modeled as an SPM, but with different access time. It's noteworthy that for a shared page, we keep the page entry only for one of the threads sharing that page. If this model changes, the mapping formulation will be slightly different. The output is a mapping of each application's virtual page to one of the physical SPMs or to the off-chip memory, with the goal of minimizing the overall memory access latency. We assume that application threads are already mapped to tiles.

The SPMPool mapping optimization problem can be solved using Integer Linear Programming (ILP). We observe that the NP-hard Knapsack Problem is reducible to the SPMPool Mapping Problem. Therefore, SPMPool Mapping Problem is NP-hard and cannot be solved in polynomial time. We present below the mapping between Knapsack problem and SPMPool Mapping:

Assuming that the platform has C cores/SPMs, overall N pages fit in on-chip SPMs, and task mapping is known (each application is mapped to a core), we want to create a mapping between Knapsack problem and SPMPool Memory mapping problem. Consider Application A is mapped to core X. If page P of this application maps to SPM i, cost of one access from core X will be $Cost(A, i)$ based on the distance to i; and overall cost will be $(Access(A, P) \times Cost(A, i))$. Page P can be mapped to any of the C SPMs in the platform. To map this situation to Knapsack problem, we can consider each of the C mappings, one item with weight 1 and Value of $1/(Access(A, P) \times Cost(A, i))$. If we create these items for every page of every application, we will have items with weight 1 and different costs. Now, SPMPool memory mapping problem can be transformed to finding the set of items that can fit in a knapsack with weight N and has maximum value (minimum cost) which is called the knapsack problem.

For the sake of completeness, we implemented this exact formulation as an ILP and ran it

with the CPLEX solver [67]. The ILP formulation generated results in the order of seconds for very small configurations (less than 16 cores) but saw an exponential increase in runtime with larger numbers of SPMs. The ILP solver ran out of memory for configurations larger than 49 SPMs.

Optimization Problem 1: SPMPool Mapping Problem

Inputs :

APP set of executing application threads
SPM set of SPMs
PAGE set of virtual pages for each thread
Cap(i) capacity (number of pages) in SPM(i)
Cost(i,j) cost of access from APP(i) to SPM(j)
Access(i, j) number of accesses to APP(i).PAGE(j) over the execution of APP(i)

Output:

$\forall a \in APP, \forall p \in a.PAGE, \forall s \in SPM$:

$$Map(a,p,s) = \begin{cases} 1, & \text{page } p \text{ in app } a \text{ allocated to spm } s \\ 0, & \text{otherwise} \end{cases}$$

1

$$\text{minimize} \quad \sum_{a \in APP, p \in PAGE, s \in SPM} Map(a,p,s) * cost(a,s) * access(a,p) \tag{2.1}$$

$$\text{subject to} \quad \forall a \in APP, \forall p \in a.PAGE : \sum_{s \in SPM} Map(a,p,s) = 1 \tag{2.2}$$

$$\forall s \in SPM : \sum_{a \in APP, p \in a.PAGE} Map(a,p,s) \leq Cap(s) \tag{2.3}$$

(2.2) guarantees a unique mapping and (2.3) is the capacity constraint.

Since the search for an optimal memory mapping is infeasible, we developed heuristic Memory Mapping Policies to approximate the optimal mapping in polynomial time (analyzed in Section 2.5.5, with experiments in Section 2.5). Each mapping policy attempts to prioritize the pages with higher need for memory resources and consists of two phases:

1. Phase A – Deciding if a page should remain on-chip or go off-chip.
2. Phase B – Mapping virtual pages to physical memory locations.

These two phases encompass the entire memory placement decision process. We now describe

three policies: Most-accessed, Neighborhood, and Local-only. In all of these policies, input, output and goal is same as SPMPool Mapping problem:

Most-accessed Policy mapping attempts to maintain the most accessed set of pages (across all executing applications) within the Pool-of-SPMs. Based on the memory access weights assigned to each page, all executing applications' pages are compared, and the most accessed overall are placed on-chip as close as possible to their home tile.

In Phase A (Algorithm 2), all threads' working sets of pages are combined and sorted based on their access weight, resulting in a sorted page list of all pages in use by executing applications. We subsequently split this list into two sub-lists: the first N pages make up the on-chip list, where N is the total number of on-chip SPM pages; all remaining pages make up the off-chip list.

In Phase B (Algorithm 3), the pages in the on-chip list are assigned to SPMs: each page in descending order of weight is assigned to the free SPM page nearest its home core (the tile on which its owner thread is executing). All remaining pages (the off-chip list) are placed in off-chip memory. For each core, memory manager keeps an adjacency list: list of all other SPMs sorted based on the distance to those SPMs (how many hops are between them).

In the case of high core-contention, it is possible for the core to enter a fragmented state where some applications' memory is far from the home core. Our experiments in Section 2.5.2 show that even in this case, SPMPool improves over the baseline due to on-chip memory

ALGORITHM 2: Most-Accessed Policy – Phase A

```
1 if thread_start then
2     sorted_page_list = Merge(Sorted_page_list, new_pages);
3 if thread_stop then
4     sorted_page_list = delete(Sorted_page_list, old_pages);
5 onchip_list = sorted_page_list[0..N-1];
6 offchip_list = sorted_page_list [N..end];
```

accesses being generally less costly than off-chip accesses.

Neighborhood Policy mapping attempts to place an application's pages with high memory access weights in SPMs as close as possible to its home tile by first claiming all of the pages in its local SPM, and subsequently searching outward for free SPM space in the "neighborhood" of surrounding tiles (Algorithm 4).

Phase A and B are combined in this policy. For each page in the incoming application's working set, if replacing an existing page would lower the overall access latency, the new page is assigned. If unable to place the new page, the search space for remote SPM allocation is progressively increased. This search for a destination is done in groups of SPMs defined as having an equal hop distance from the home tile. Once the search neighborhood is expanded past a threshold without an assignment, the pending page and all of the entering application's remaining unmapped pages are relegated to off-chip memory.

Local-only Policy mapping (Algorithm 5) is the baseline that only allows each thread to use its local SPM. All pages in a thread are sorted based on their weight and the top *Cap(i)* pages are mapped to the local SPM while the other pages are mapped to off-chip memory. *Cap(i)* is the number of available pages in each SPM. This emulates policies that do not support remote SPM utilization. As a reminder, for each shared page, there is only one entry in one of the threads sharing that page.

ALGORITHM 3: Most-accessed Policy – Phase B

```
1 for page in sorted_page_list do
2     home = home_core(page);
3     for SPM in home.adjacency_list do
4         if SPM.has_free_slot() then
5             SPM.assign(page);
6             SPM.reduce_free_slots();
7             break;
```

2.4.2 SPMPool Architectural Assists

Recall from Figure 2.1 that the pooling of on-chip memory resources requires hardware and software architectural assists within each tile. To provide each application a virtual private address space (Figure 2.6a) with data that may be physically distributed throughout the on-chip network, SPMPool must allow the applications to both address and access remote

ALGORITHM 4: Neighborhood Policy

```
1  hop_distance = 1;
2  for page in sorted_page_list do
3      home = home_core(page);
4      for SPM in home.adjacency_list do
5          for SPM in home.SPM_list(hop_distance) do
6              if SPM.has_free_slot() then
7                  SPM.assign(page);
8                  break;
9              else
10                 hop_distance++;
11         for SPM in home.SPM_list(hop_distance) do
12             if SPM.min_page_importance <page.importance then
13                 SPM.evict(SPM.page_min_importance);
14                 SPM.assign(page);
15                 SPM.update_min_page_importance();
16                 break;
17         if hop_distance >threshold then
18             break;
19     assign any remaining pages to main memory;
```

ALGORITHM 5: Local-only Policy

```
1  if thread_start then
2      onchip_list = thread.sorted_pages[0..S-1];
3      offchip_list = thread.sorted_pages[S..end];
4      SPM = thread.local_SPM;
5      for page in onchip_list do
6          SPM.assign(page);
7  if thread_stop then
8      thread.local_SPM.free();
```

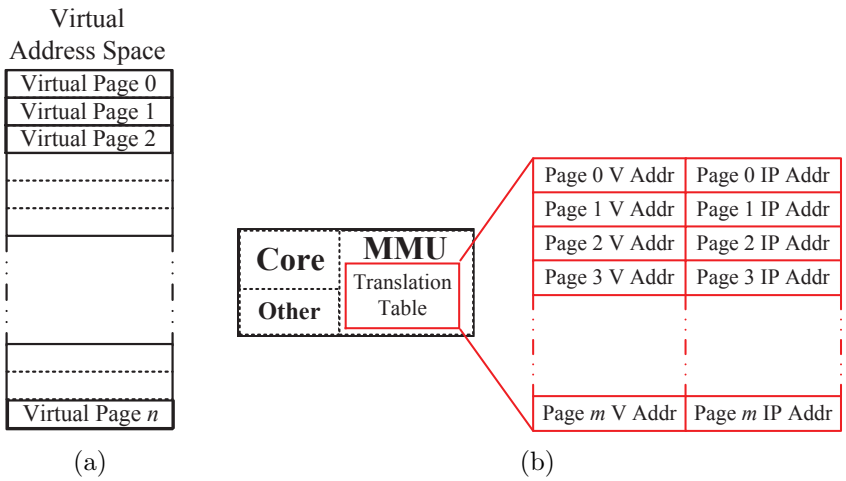


Figure 2.6: (a) Virtual address space of a single application (b) SPMPool-specific translation table on each tile

SPMs, and store information about the physical location of their data.

Each tile has an augmented translation table that contains SPMPool mapping information. These SPMPool-specific translation tables specify virtual address to physical on-chip SPM address mappings for any of the application's pages that are stored on-chip (Figure 2.6b). In multi-threaded applications, all the cores running different threads of the same application should have an entry for the shared pages (if those pages are mapped on-chip). Any memory access initially checks this translation table for the desired page. If an entry for the page exists, a local or remote SPM access is initiated depending on the mapping indicated. If an application's page is not in its local translation table, the memory access defaults to a standard main memory access. For a remote SPM access, the virtual memory management generates an intermediate physical address (IPA) for each SPM page. In this scheme, the SPM IPAs (held in the translation tables) can be decoded to determine where the specific SPM page resides physically on chip. The MMU on each tile contains hardware to handle direct SPM access requests to and from remote tiles, along with a TLB to handle standard page access to main memory. Figure 2.7 illustrates a sample remote SPM access by an executing application.

The application executing on Core 0 initiates a memory load by sending the request to its local MMU, which contains the translation table. In this case, based on the virtual to IPA translation of the requested address, the MMU determines that the data resides on the remote SPM located on Tile 1. It sends a request over the NoC for the data, and that request is received and serviced directly by the MMU on Tile 1 (red path in Figure 2.7). The data is returned to the requesting core through the same path (green path in Figure 2.7).

2.5 Experimental Setup and Results

2.5.1 Experimental Setup

The SPMPool simulation infrastructure is designed to evaluate overall memory access latency of user-specified workloads on a variety of many-core configurations. We developed a Java-based simulator that implements the SPMPool runtime manager routine and calculates the memory access latency of the workload. The simulator takes as input the memory traces of all applications in the workload along with the access weight values of each application's working set. The simulator models the flow of the SPMPool Memory Manager in Figure 2.5 and tracks all memory accesses throughout execution of the workload. For our experiments, we used a variety of benchmarks from SPEC2006 [62], MiBench [55] and PARSEC [17] suites that have varying memory-use characteristics, and combined them to create diverse

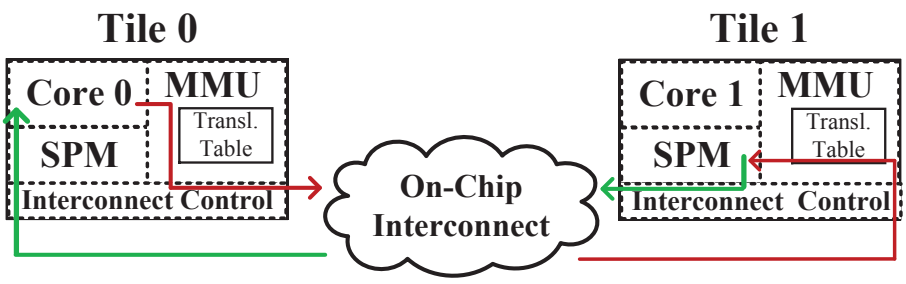


Figure 2.7: Example remote SPM access.

Table 2.2: Cost of Access in 4x4 platform

SPM #:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Off-chip
Core 0	1	6	9	12	7	9	11	14	9	12	13	16	11	13	16	18	90
Core 1	6	1	7	9	9	7	9	11	11	9	12	14	14	11	14	17	90
Core 2	9	7	1	7	11	9	7	9	14	11	9	12	16	14	11	14	90
Core 3	12	9	6	1	13	10	9	7	15	13	11	9	17	16	13	11	90
Core 4	7	8	11	14	1	6	9	11	7	9	12	14	9	11	14	16	90
Core 5	9	7	9	11	7	1	6	9	9	7	9	12	12	8	11	13	97
Core 6	12	9	6	9	9	7	1	7	12	10	7	9	13	11	9	11	96
Core 7	13	11	8	7	12	9	6	1	13	11	9	6	16	14	11	9	90
Core 8	8	11	13	16	7	9	11	14	1	6	9	12	6	8	11	13	90
Core 9	11	9	12	13	9	7	10	12	7	1	7	9	9	6	9	11	96
Core 10	14	11	9	12	11	9	7	9	9	7	1	7	11	9	6	9	96
Core 11	16	14	10	9	14	11	9	7	11	9	6	1	13	11	9	7	90
Core 12	11	13	16	18	9	11	14	16	7	9	12	14	1	6	9	11	90
Core 13	13	11	14	16	12	9	12	14	10	7	9	12	7	1	7	9	90
Core 14	16	13	11	14	13	12	9	11	11	9	7	9	9	7	1	7	90
Core 15	18	16	13	11	15	14	11	9	13	11	9	7	12	10	6	1	90

workloads. To generate each workload, benchmarks and their associated entry time were selected randomly, all running to completion (the details are described in the following sections). Memory traces are obtained by running benchmarks with gem5 simulator [19]. These memory traces are the inputs of SPMPool simulator. The traces were parsed by the SPMPool analyzer to determine the memory access weight of each page for every application. For this set of experiments the memory access weight is the number of times a page is accessed by its application.

To calculate the latency of each memory access, we considered the time required to communicate between the source and destination tiles and the time required to access the memory bank. Noxim [47] was used to statically obtain the NoC communication delay per-hop accounting for network contention using random-injection. CACTI [92] was used to obtain memory bank access latency. The cost of access from each core to each SPM (number of cycles) is shown in Table 2.2. These cost numbers are obtained by adding the numbers extracted from Noxim and CACTI. Each tile is associated with a 64KB SPM. All virtual and

Table 2.3: List of Benchmarks

Application Type	Combinatorial Optimization	Artificial Intelligence	Video Compression	Image Ray-tracing
Benchmark	mcf	gobmk	h264ref	povray
Number of different input sets	1	7	3	1

physical memory is divided into 4KB pages. There are no caches present, and we assume all data and instructions are stored in RAM. We conservatively assumed a bandwidth of 8GB/s for on-chip communication and 160 MB/s for accessing main memory to calculate overheads (real platforms have higher bandwidth e.g., Intel SSC supports 64GB/s [64]). Simultaneous accesses to one memory bank is not considered in our simulations.

2.5.2 Experimental Results - X86

Table 2.3 lists the benchmarks and number of different sets of inputs used in our experiments. To generate each workload, benchmarks and their associated entry time were selected randomly, all running to completion. Memory traces were extracted by executing the benchmarks on x86 architecture. We extracted the traces from gem5 simulator [19] while executing the benchmark.

Figure 2.8 depicts the dynamic memory characteristics of the benchmarks. The graphs show the number of accesses to the highest ranked pages of a sample execution. The varied access patterns of these benchmarks make them good candidates for SPMPool working set. Also it is shown by Jaleel [73] that about 50 % of all instructions in these SPEC benchmarks are memory accesses. Hence, reducing memory access latency leads to considerable improvement in execution time.

In the following, we discuss three sets of experiments. The first set shows the memory access latency improvement for different utilization points of the same configuration. In the

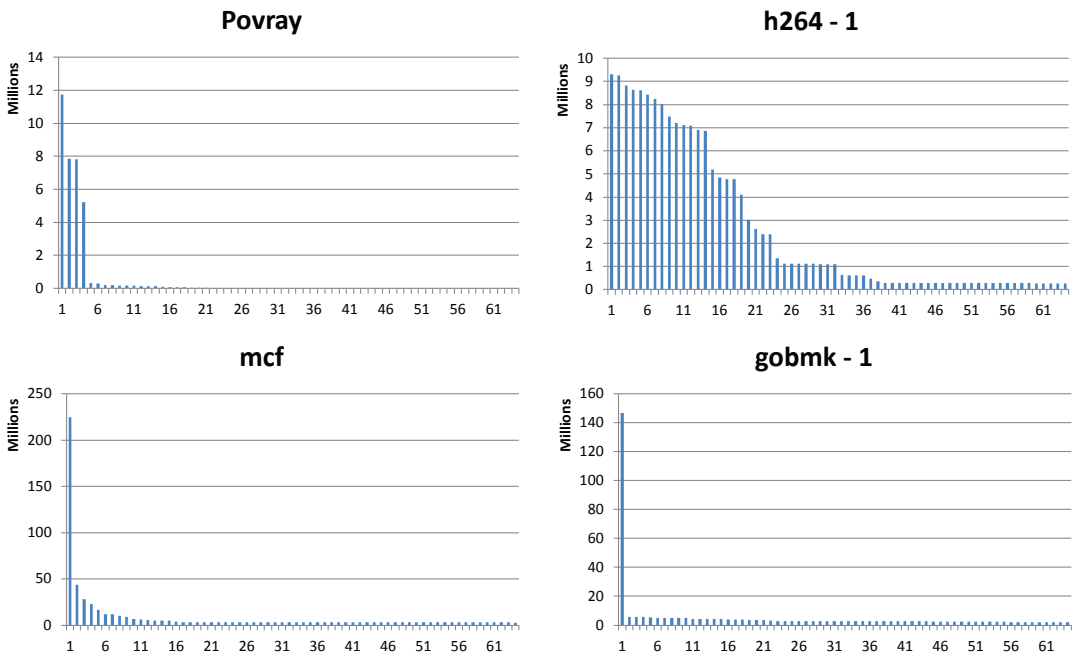


Figure 2.8: Highest accessed pages of benchmarks used in the experiments

second set, the SPMPool Memory Mapping Policies are studied. In the third set, workload dependence of SPMPool is studied. Every simulation shown is run with a unique randomly generated workload.

Latency improvement for different utilizations

Figure 2.9 shows the improvement in overall memory access latency achieved by the SPM-Pool most-accessed placement policy over the local-only policy for both 4x4 (16 cores) and 8x8 (64 cores) configurations at different utilization points. Utilization is the percentage of concurrently active cores throughout runtime. At low utilization points ($\leq$25%), SPMPool reduces memory access latency by more than 35% for both 4x4 and 8x8 configurations. As the number of utilized cores increases, the advantage of SPMPool over non-sharing organizations decreases. However, we still observe about 7% reduction in overall memory latency when the 8x8 configuration is 100% utilized. In any fixed utilization, for different sets of

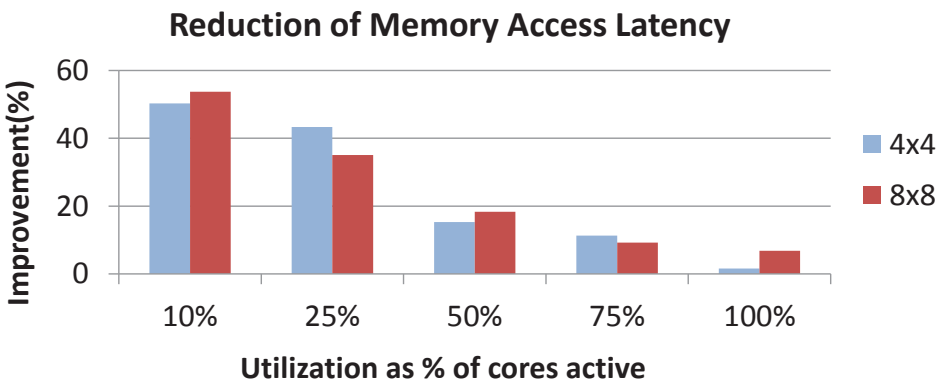


Figure 2.9: Percentage of overall memory access time improvement achieved by the SPMPool most-accessed placement policy over the local-only policy.

applications and scheduling, we observe a significant variation (up to 25%) between memory access latency improvements, so we don't necessarily observe a consistent relationship between configurations through the different utilizations. For this set of experiments, SPMPool reduces memory access latency by up to 55% at the lowest utilization points by sharing the vast amount of unutilized SPM resources.

Policy comparison for different configurations

Figure 2.10 includes a comprehensive comparison of the SPM mapping policies introduced in this chapter, for different utilizations on configurations from 4x4 (16 cores) to 16x16 (256 cores). One additional placement policy studied is a simplified version of the most-accessed policy. The most-accessed policy re-maps all on-chip memory pages every time it generates a memory map, which can induce excess memory migration overhead. In the *simplified most-accessed policy*, memory migration is limited to between on- and off-chip memory only, not between SPMs.

To be more specific, if a page is currently assigned to one of the SPMs, and the new memory mapping changes its location, we move that page only if it is mapped to off-chip memory in the new mapping. By this restriction, we can reduce the overhead of memory

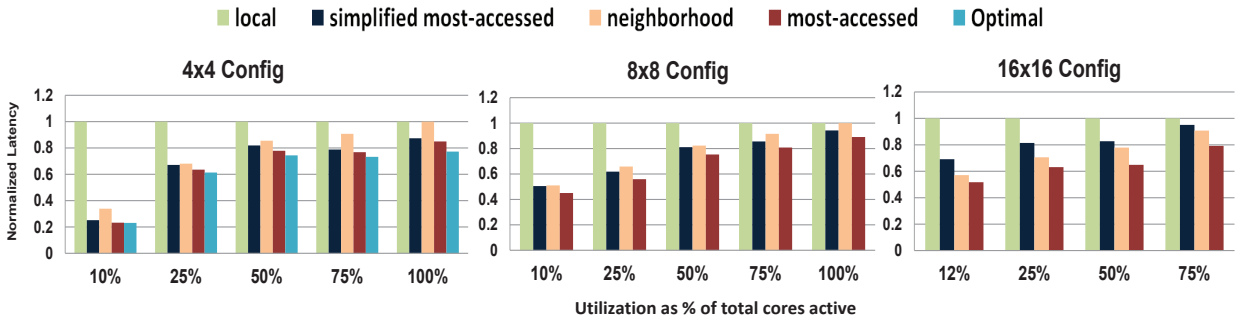


Figure 2.10: Scalability analysis by comparing total memory access latency normalized to local-only policy

migration significantly. The range of utilizations simulated for 4x4 and 8x8 consisted of five points from 10% to 100%. The 16x16 configuration was run for simulations from 12% to 75% utilization. The threshold for neighborhood policy used was 10 hops, as we did not observe significant improvement beyond this point. For efficiency we restrict migration to between on- and off-chip.

We observe in Figure 2.10 that the overall memory access latency is reduced in all cases over the local policy. The most-accessed policy yields the greatest reduction, but all SPMPool Policies are able to reduce latency for all configuration sizes from 4x4 to 16x16, even at high utilizations ($\geq$50%). Also, given the trends in dark silicon [44], we can expect that 100% utilization may not be a common use-case for beyond hundreds of cores[3]: hence our SPMPool strategies can be expected to yield significant performance improvement, especially when platform resources are not completely utilized. In the best case SPMPool reduces memory access latency by 76%, 55%, and 48% for 4x4, 8x8, and 16x16 configurations respectively at ˜10% utilization.

Relative performance between SPMPool Policies changes as configuration size grows to hundreds of cores (Figure 2.10). For 4x4 and 8x8 configurations, both versions of the most-

[3]Replacing cores with memory doesn't solve the dark silicon problem. Although we cannot turn on all cores simultaneously, they are useful for thermal management and aging reduction using activity migration and other techniques

accessed policy outperform the neighborhood policy. However, as configuration size grows, the severity of across-chip remote SPM access penalty increases, and as a result the neighborhood policy outperforms the simplified most-accessed policy for the 16x16 configuration. SPMPool shows promise as configuration scales to hundreds of cores: for 16x16 at 50% utilization we observed a minimum of 18% latency reduction.

Workload Dependence

In Section 2.5.2 interdependency of system utilization and performance is studied. But the SPMPool performance is not only dependent on the utilization, it is also highly related to the workload (i.e., set of concurrently executing applications and their inputs) and how it is scheduled. Even if the utilization is kept constant, a change in workload may significantly impact memory requirements, and subsequently the performance of SPMPool. In our experiments we observed a significant variation (up to 25%) between memory access latency improvements for different workloads (for a fixed utilization). Figure 2.11 shows the maximum latency improvement variation for different workloads.

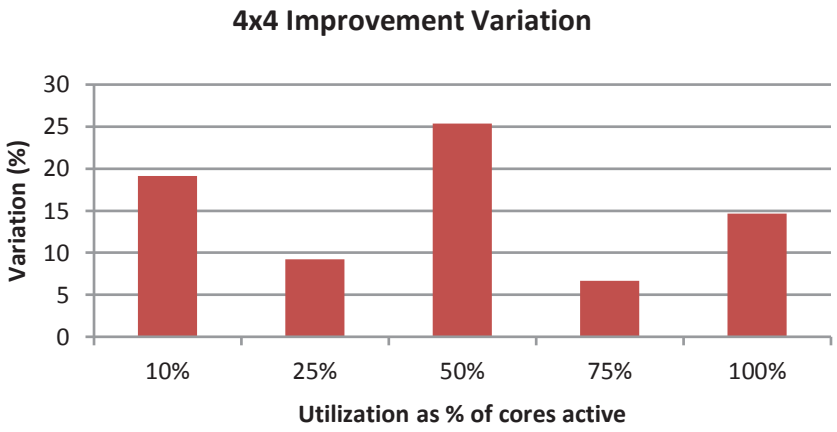


Figure 2.11: Maximum improvement Variation (%) between sets of applications for 4x4 configurations.

Table 2.4: Off-chip memory access ratio of most-accessed policy versus local-only policy

	Memory Access Ratio	
Utilization	4x4 Config	8x8 Config
10%	0.142	0.232
25%	0.596	0.635
50%	0.752	0.778
75%	0.728	0.768
100%	0.817	0.849

Power consumption

Access to off-chip memory is an order of magnitude more costly than on-chip accesses in terms of power consumption. Although SPMPool Policies primarily target the reduction of latency as the main goal, they can also reduce the number of off-chip memory accesses significantly. Consequently we estimate that using SPMPool improves system power consumption. Table 2.4 shows the ratio for off-chip accesses of the most-accessed policy versus the local-only policy, demonstrating the potential for concomitant power reduction.

2.5.3 Experimental Results - ARM

Because SPMPool targets embedded systems, we must explore its effects on appropriate architectures and workloads. Therefore, we selected different benchmarks from MiBench [55] – an embedded benchmark suite – to execute on an ARM architecture configuration. The selected benchmarks represent different embedded application domains. Table 3.1 lists the benchmarks and number of different inputs used in this set of experiments. Memory traces were extracted by executing the benchmarks with gem5 simulator on ARM architecture. The set of benchmarks and their entry time were selected randomly.

Figure 2.12 illustrates the improvement in overall memory access latency for both 4x4 (16 cores) and 8x8 (64 cores) configurations at different utilization points. We compared the

Table 2.5: List of benchmarks used from MiBench suite

Application Type	Benchmark	Number of inputs	Application Type	Benchmark	Number of inputs
Consumer	jpeg	2	Network	dijkstra	2
Consumer	typeset	2	Network	patricia	2
Consumer	lame	2	Telecomm	fft	2
Automotive	qsort	2	Telecomm	gsm	2
Automotive	bitcount	2	Security	blowfish	2
Automotive	susan	2	Office	stringsearch	2

most-accessed placement policy to the *local-only* policy baseline. The results are compliant with the results on X86 architecture, illustrated in Section 2.5.2. SPMPool can reduce the memory access latency significantly in low utilization points (up to 72%). As the number of concurrently executing tasks increases, the increased resource conflicts between different tasks reduces the benefits of SPMPool.

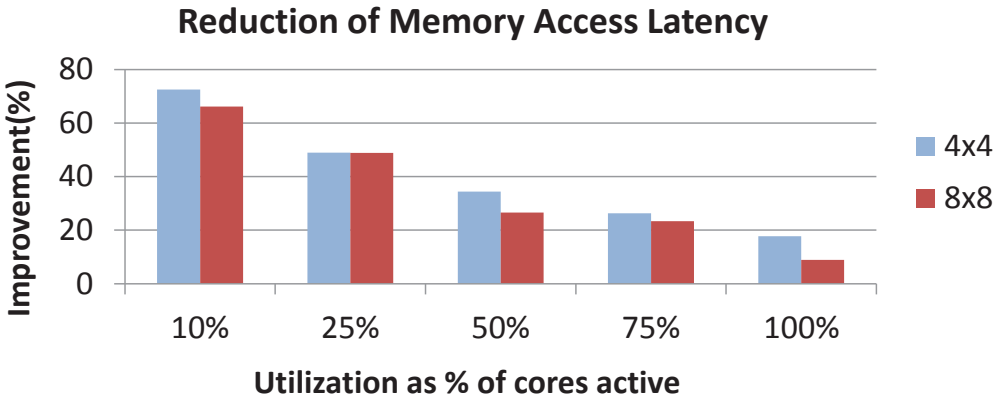


Figure 2.12: Percentage of overall memory access latency improvement of SPMPool (most-accessed policy) for MiBench benchmarks on ARM architecture over the local-only policy.

2.5.4 Experimental Results for Multi-threaded applications

SPMPool supports multi-threaded applications with shared data (details in Section 2.4). In lieu of implementing a coherence protocol in our memory architecture, SPMPool does not support data duplication of shared memory. Instead, mechanisms are included to allow access by multiple threads to a single copy of a shared piece of data. To exercise this use

Table 2.6: List of benchmarks used from PARSEC suite

Benchmark	**Number of inputs**	**Benchmark**	**Number of inputs**
blackscholes	2	canneal	2
bodytrack	2	x264	2
facesim	2	ferret	2
fluidanimate	2	freqmine	2
streamcluster	2	swaptions	2

case, we ran a set of experiments with multi-threaded workloads. The baseline policy forced all shared data to remain in off-chip memory. Unlike previously discussed experiments, the purpose of these experiments was primarily to demonstrate functionality, not quantify improvements.

PARSEC, a multi-threaded benchmark suite, was used for creating the workload. Table 2.6 lists the benchmarks and number of different sets of inputs used in these experiments. To generate each workload, benchmarks and their associated entry time were selected randomly, all running to completion. Memory traces were extracted by executing the benchmarks with gem5 simulator, each running 4 threads. Due to limitations in our simulator, we did not simulate locking mechanisms for accessing the shared memory.

The overall memory access latency of SPMPool is compared to the baseline (shared data in main memory) in Figures 2.13 and 2.14 for both 4x4 and 8x8 configurations at different utilization points. Using a mechanism which enables bringing shared data to on-chip memory can decrease the memory access latency significantly. In low utilization points, SPMPool could achieve up to 86% improvement in memory access latency. In high utilization points, this improvement is still compelling (more than 60%). As keeping coherency of data is a major obstacle in multi and many-core systems, these results show the benefits of SPMPool in systems without any coherency protocol.

2.5.5 Overhead

The sources of overhead imposed by SPMPool include communication, computation, storage, and memory migration. Frequency of change in the system impacts the overhead; more SPMPool events lead to more overhead. Architectural assists added to each core impose some hardware overhead in the form of storage. Translations of virtual addresses to intermediate physical addresses, and memory address weights must be stored on each core for its locally executing application There is no other explicit added hardware in SPMPool. The SPMPool Memory Manager incurs communication overhead due to the need to send memory mapping updates to affected cores. These cores will halt the execution of their applications until translation tables are updated and memory migration is completed. In this section, the different overhead components of SPMPool are analyzed, demonstrating that SPMPool has a relatively low overhead. Note that even without SPMPool, some book-keeping and memory migration is inevitable for any SPM-based system.

Communication and Computation Overhead

The SPMPool runtime manager communicates with cores to generate a new memory map. On application start, the application's memory access weights are sent to the Manager. On

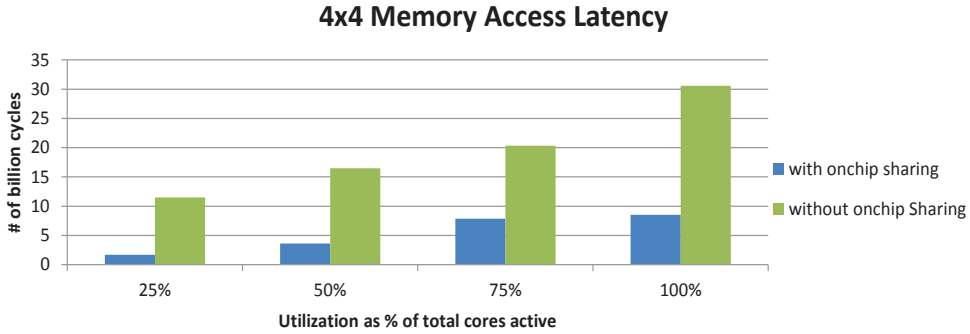


Figure 2.13: comparison of memory access latency between SPMPool and baseline (shared data in main memory) for 4x4 platform

application stop, the application sends a message to the Manager to indicate its completion. After creating a new memory map, the Manager sends messages to update the translation table on each affected tile. Assuming N is the page capacity of all SPMs, the manager needs to send and receive at most $2*N$ messages on each event, which grows linearly with the configuration size.

We consider the *most-accessed* mapping policy in Section 2.4.1 for estimating the overhead of updating the memory map (the overhead of other mapping policies can be obtained similarly). The mapping policy first decides if a page should remain on-chip or go off-chip. This involves merging pre-sorted lists which can be done in O(N) = O(#SPMs * #pages-per-SPM). Next, it maps virtual pages to physical memory locations. This involves traversing the list of all SPMs for each page and can be done in O(N * #SPMs) = O(#SPMs * #pages-per-SPM * #SPMs). As #pages-per-SPM is a constant number, the computational complexity of the mapping algorithm is O(#SPMs^2). All overhead incurred by the SPMPool runtime manager to this point does not require any applications to suspend execution and therefore has a low impact on the overhead of the entire system.

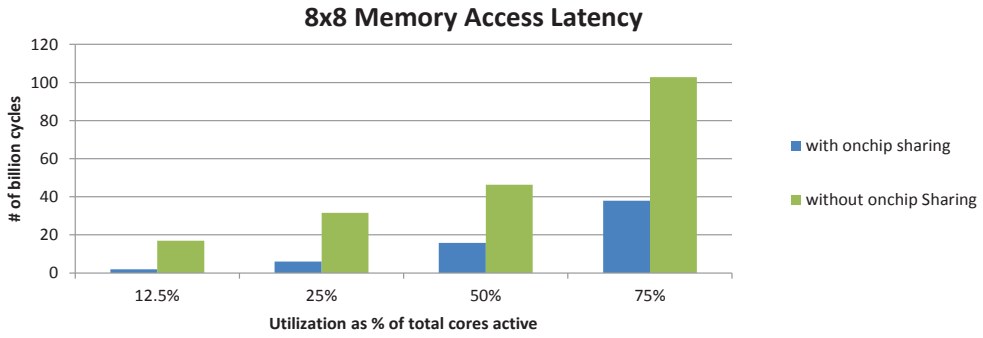


Figure 2.14: Comparison of memory access latency between SPMPool and baseline (shared data in main memory) for 8x8 platform

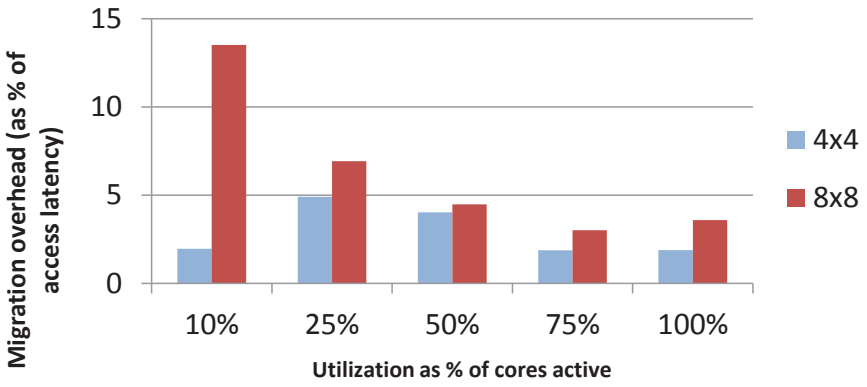


Figure 2.15: Memory Migration Overhead

Storage Overhead

To maintain the virtual to physical memory mapping and efficiently execute the mapping algorithm, the Manager and each tile need to hold some SPMPool-specific information. If the total capacity of SPMPool is N pages, the manager needs to keep the number of accesses for #apps * N pages. To locate its data in physical memory, each tile needs to have a translation table for any on-chip pages belonging to the application executing on its core. This table is also necessary for non-SPMPool organizations.

For instance, in an 8x8 configuration with the characteristics mentioned in Section 2.5, N is 64*16=1024. If each page entry in the runtime manager needs 20 bytes to store the memory access weight, the runtime manager needs 64*20*1024 = 1.3MB of storage. 10KB of data is sufficient for each tile to maintain its translation table.

Memory Migration Overhead

After generating a new memory map, some application pages may need to be moved between off-chip and on-chip memory, as well as between SPMs. During this migration, affected applications should halt their execution. Even in non-SPMPool organizations, some degree of memory migration is inevitable. Due to the fact that SPMs are typically small (64KB

in this work), and interconnection networks are relatively high bandwidth, overhead due to memory migration is low. We monitored the memory migration in our simulator for the experiments described in Section 2.5. We computed the overall memory migration latency and compared it to the total simulated execution time to estimate the memory migration overhead. Our results, shown in Figure 2.15, show that for 4x4 and 8x8 configurations the memory migration overhead is relatively small for most cases – in the range of 2 percent to 6 percent considering the most-accessed policy. Overhead for the 8x8 configuration is slightly higher than 4x4 configuration. For example, if we consider 25% utilization in 4x4 configuration, 1.44 billion cycles are spent for memory accesses with an overhead of 70 million cycles. In the same utilization point, the memory access latency and overhead for 8x8 configuration are 7.71B and 370M cycles respectively. This trend continues to systems with configuration size of 16x16. The overhead increases with system configuration size due to longer communication paths and more memory migrations.

If we account for all types of overhead outlined above, SPMPool incurs a relatively small burden in terms of time and memory overhead for systems sized at a couple of hundred cores.

2.6 Discussion

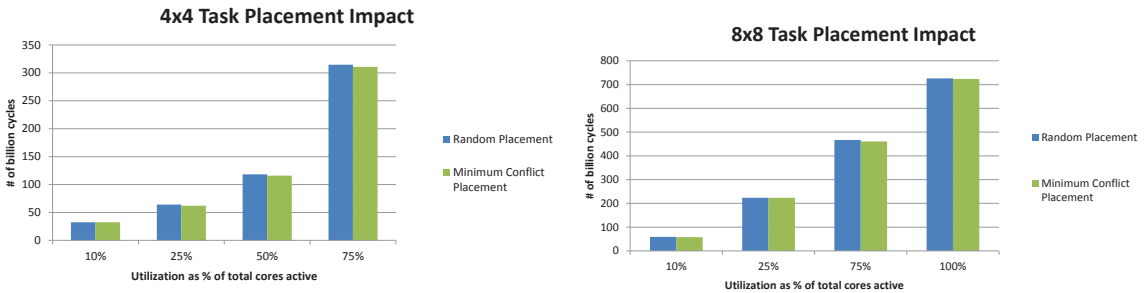
2.6.1 Scalability and multi-agent management

As the number of cores reaches 256, the scalability of the current version of SPMPool becomes an issue – an on-chip memory access may be more costly than an off-chip access in the most extreme cases. Multiple pools become necessary due to severe delay accessing remote tiles across chip. A single centralized runtime manager is no longer sufficient and we must move to a distributed management topology. This opens up new challenges in managing distributed pools of SPMs which will be discussed in Chapter 3.

2.6.2 Sensitivity to Application Mapping

Relative assignment of applications to cores in NUMA many-cores affects the memory access latency of the system. For all experiments discussed to this point, we assumed each application upon dispatch is randomly mapped to an available core without any knowledge of current programs executing. To quantify the degree of impact application placement has on a memory placement technique such as SPMPool, we re-evaluated a subset of the previous experiments while employing an alternative application mapping strategy. This placement strategy searches for a chip region with memories that are accessed least by existing cores. The SPMs in this region potentially have the minimum amount of conflicts with other cores in terms of memory accesses, and contain the least critical data on-chip for existing applications.

All the experiments have been performed for moderately sized many-cores (less than 100 cores) and as shown in Figures 2.16a and 2.16b, the impact of using an intelligent task placement policy, such as our minimum conflict policy, on overall memory access latency is negligible for any memory placement strategy discussed in this chapter. Large sized many-cores should be studied separately for the effects of task mapping tied with memory mapping on the overall SPMPool performance.



(a) (b)

Figure 2.16: Memory access latency for the same workload using different task placement policies.

2.7 Conclusion

In this chapter, we presented SPMPool, a strategy for sharing SPMs across multiple simultaneously executing applications on a many-core platform. To the best of our knowledge, SPMPool is the first approach to provide dynamic runtime memory management for unpredictable workload supporting SPM sharing. SPMPool reacts to unpredictable workloads by dynamically updating the memory mapping at runtime as the set of concurrently executing applications changes. We demonstrated that SPMPool can achieve 48% reduction in overall memory access latency for a 256-core platform by sharing underutilized SPM resources among concurrently executing applications. Analysis of our implementation indicated that the observed SPMPool benefits could be achieved at a relatively low overhead.

In a platform with hundreds of cores, a central memory management is very inefficient. In Chapter 3, we propose different schemes to manage platforms with hundreds of cores in a distributed manner. Also in this chapter, we assumed that applications are profiled offline. Moreover, SPM re-mapping routine is only triggered when applications start or stop their execution. These assumptions help to introduce the concept of SPMPool, but are not realistic or efficient in real systems. In Chapter 4, we propose an online memory phase detection scheme in order to increase the adaptivity of memory management to application changes which also addresses the quantification of memory requirements without offline profiling.

Chapter 3

Auction-Based Memory Mapping in Many-core Systems

3.1 Introduction

Growing computing resources on a chip is commonly exploited in the form of multi and many-core platforms and the increase in the number of cores is projected to continue for the near future (Chapter 1). Many-core platforms provide the opportunity to execute a dynamic and unpredictable workload where many applications –with different resource requirements– can be executed concurrently.

Concurrently running applications compete for the available resources. A competent run-time management mechanism is required to fulfill applications requirements, efficiently use resources, increase the performance of the entire system, meet power and thermal requirements, and resolve the conflicts between competing applications. The resource management problem becomes more complicated in many-core platforms, due to higher number of resources and competing applications.

A central management scheme can govern a small platform with a few number of cores efficiently [18, 96, 49, 103]. When platform size approaches hundreds of cores, a central manager becomes incapable of administering the entire system with a reasonable overhead due to:

- Increased communication distance: more time and energy is needed to send data to or receive it from the manager.
- Network congestion and creation of a hotspot: every sensing, monitoring, or command is directed to or originates from the manager. This creates a thermal and communication hotspot.
- Increased storage requirement: for bookkeeping of applications and their data, the manager needs to store all applications' information which increases excessively with high number of resources and applications.
- Increased computation: the number of computations for the resource assignment problem is dependent on the size of problem and imposes a huge overhead in presence of hundreds of cores and applications.
- Single point of failure: the manager's failure can stop the entire system from functioning which is an important dependability concern.

Therefore, using a central manager degrades system performance significantly and makes any management system unscalable. To solve this problem, a distributed management scheme should be employed to increase the efficiency of resource management in many-core systems.

Several approaches have been introduced to govern many-core platforms in a distributed manner. In most of these approaches, the entire platform is divided into different clusters and an agent manages each cluster. These clusters might be fixed-size or determined by a central manager at arrival time of applications ([4, 5]) or might expand and shrink based on

completely distributed mechanisms through local communication mechanisms ([46, 34, 79, 129]). The majority of proposed approaches only target task to core mapping.

Some management systems use market-based algorithms to tackle resource assignment problems. In these systems, some agents try to maximize their benefit by obtaining resources (objects) of the system. Auction is one of the oldest mechanisms used in trading and commerce. It has been shown that exploiting principles of an auction mechanism is a successful approach to assigning resources to agents [77, 14, 16, 15, 132, 86, 42, 45, 88, 137, 87, 134, 35]). Most required computations for auction-based assignment mechanisms can be performed in parallel and executed with central or distributed management – with or without shared memory.

We have proposed SPMPool (Chapter 2) as a platform to share SPM resources in many-core platforms. In SPMPool, memory resources are assigned to applications based on applications' requirements. Performance of SPMPool is dependent on the efficiency of memory mapping policies and overhead of memory management. The management scheme introduced in Chapter 2 is a central scheme and has the above-mentioned limitations when moving toward hundreds of cores. Thus, using a distributed management for SPMPool is inevitable to manage the resulting complexity.

The efficiency and flexibility that auction based algorithms provide make them good candidates for memory mapping in many-core systems. In this chapter, we model the memory mapping problem of SPMPool using an auction mechanism and propose a two layer management approach for platforms with hundreds of cores. In this approach, the platform is divided into different pools and each pool has a local manager that acts as an agent. Agents use the distributed auction mechanism to claim the resources in other pools.

The rest of this chapter is organized as follows: Section 3.2 reviews the existing distributed resource management schemes and auction based resource assignment methods. Section 3.3

outlines the memory mapping problem of SPMPool as an auction mechanism. In Section 3.4, a two layer distributed approach for managing SPMPool is proposed. Experimental setup and results are discussed in Section 3.5.

3.2 Related Work

Runtime resource management of multi and many-core architectures becomes very challenging in the face of increasing available resources and abundant number of applications with variable requirements. To fulfill the growing demand for adaptivity, numerous runtime management schemes have been proposed which can be categorized in two general class of central and distributed management. Central management schemes are effective for relatively small platforms. The ACTORS approach [18] allocates virtual platforms to each application and the manager maps virtual resources to available physical resources. Nollet et al. [96] used A runtime task migration based on an NOC resource management heuristic. A feedback control and optimization is used by Fu et al. [49] to minimize the power consumption by monitoring CPU utilization of cores and applying core-level DVFS. Sabin et al. [103] used an iterative approach to assign proper number of cores to parallel jobs.

A centralized scheme is incapable of managing many-core platforms with hundreds of cores. Accordingly, several distributed schemes have been proposed to manage many-core platforms. Anagnostopoulos et al. [4, 5] propose a central manager which analyzes incoming applications and dispatches them to local managers. An agent based distributed mapping, called ADAM, is used by Al Faruque et al. [46]. It implements a cluster negotiation algorithm to form virtual clusters at runtime and a heuristic algorithm is in charge of task mapping inside each cluster. An enhanced re-clustering technique for task mapping is proposed by Cui et al. [34].

In the invasive computing paradigm [59] an application may dynamically expand on parallel cores or retreat based on the possible/required parallelism. A runtime management for this paradigm, called DistRM, is introduced by Kobbe et al. [79]. In DistRM, each agent is associated with an application and autonomously tries to increase the speedup of its application by searching for cores on the chip. A hierarchical and multi objective distributed resource management is proposed by Bellasi et al. [13]. Weichslgartner et al. [129] proposed a decentralized scheme for mapping of tree-structure application. Each application initiates mapping autonomously and each process embeds its succeeding tasks only with local view of neighbor nodes. Also different distributed thermal management schemes have been used in many-core platforms (e.g., Ge et al. [51] and Sartoti et al. [105]). To the best of our knowledge, no viable distributed solution for memory mapping has been proposed in many-core systems.

Market based resource allocation algorithms such as auction-based algorithms have been studied for many years. Auction is a basic mechanism in markets in which some bidders try to acquire some objects. If an object (with predefined initial price) is desirable for some bidders, they try to obtain the object by bidding on it and each bid increases the price of the object. This process ends when the price is so high that it's not desirable for other bidders and the highest bidder gets hold of the object. In auction based algorithms, some agents try to maximize their benefits by bidding on and obtaining the resources of the system. Different variations of auction algorithms such as open and sealed bid, ascending-bid, descending bid, first-price sealed-bid, and second-price sealed bid auction have been proposed and discussed in auction theory [77]. The initial work to use auction as a mechanism for resource assignment goes back to the late 1970s in which Bertsekas [14] proposed an auction mechanism to solve the basic assignment problem when the number of agents and bidders are equal. Bertsekas [16] also outlined parallel synchronous and asynchronous variations of this mechanism. An asynchronous auction algorithm without accessing shared memory is presented by Zavlanos et al. [132]. In this mechanism, each agent bids on objects based on the current view of the

prices. Some prices might be outdated at any particular time, but the view of the system will be updated eventually after passing the information between agents.

Auction algorithms have been used to solve assignment and optimization problems in different fields of computer science. The distributed nature of this algorithm makes it a viable solution for distributed task allocation in different domains such as wireless sensor networks ([86, 42]), peer to peer networks ([45]), and multi-robot environments ([88, 137]). Resource allocation is a critical part of cloud computing and variations of the auction mechanism has been used to optimize it (e.g., Lin et al. [87]). Zhang et al. [134] used auction mechanism for resource allocation in mobile cloud computing systems. Also it is a popular method for scheduling tasks in grids ([35, 52, 117]). Network resource allocation systems has also benefited from market-based algorithm. For instance, Sun et al. [118] used auction algorithm for wireless channel allocation.

To the best of our knowledge, ours is the first work to manage the memory mapping of a many-core system in a distributed manner, using the auction mechanism. Despite all the efforts in using auction-based resource management, it has not been exploited and evaluated in SPM mapping.

3.3 Auction Mechanism for Central Management of SPMPool

The auction algorithm is one of the important methods to solve the assignment problem. In this section we illustrate how the auction algorithm can be exploited to solve the SPM mapping problem of SPMPool, introduced in Section 2.4.1. As a reminder, SPMPool memory manager uses a central management scheme and some heuristics to solve this problem. We show how to solve the same problem more efficiently using an auction algorithm.

In a generic assignment problem, n objects are assigned to n persons (one object each). Assume object j has price p_j and person i will be benefited a_{ij} by obtaining object j; so the net benefit of obtaining object j for person i is $a_{ij} - p_j$. The objective of the assignment problem is to find a one-to-one assignment that maximizes the total benefit: $max \sum_{i=1}^{n} a_{ij_i} - p_{j_i}$.

Bertsekas [15] showed that assignment problem is equal to the *economic equilibrium problem* and formulated an auction algorithm to obtain *economic equilibrium* in which each person acts as an economic agent and tries to maximize its own benefit. We use this auction algorithm to model the SPM mapping problem.

3.3.1 SPM Mapping Problem Modeling

SPMPool Memory Mapping, introduced in Section 2.4.1, selects a subset from the numerous pages of concurrently executing applications and assigns them to the limited number of on-chip SPM slots in order to minimize overall memory access latency. In this section, we propose a solution to SPMPool memory mapping using the auction scheme.

To model memory mapping as an auction mechanism, we need to specify objects and their initial prices, bidders and their potential benefits by obtaining each object, and the auction algorithm. Figure 3.1 demonstrates the model to represent the SPM mapping in many-core systems as an auction mechanism:

1. Objects: each SPM and Main Memory page is an object. All SPM pages have the same price and all Main Memory pages have the same price at the beginning.

2. Bidders: each application page is a bidder. The number of objects should be at least equal to the number of bidders. Application pages (bidders) usually outnumber the available SPM pages; therefore, objects representing main memory pages should be

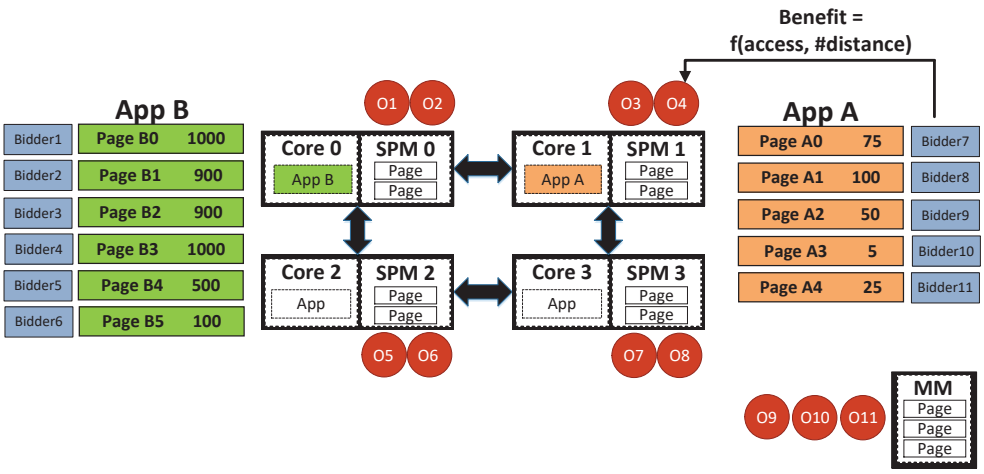


Figure 3.1: Modeling SPM mapping problem with auction mechanism

added to make the number of bidders and objects equal. The benefit of obtaining each object (memory page) should be defined based on the distance between the bidder (application page) and the object (memory pages), and also the number of accesses to the bidder.

3. Final assignment: executing the action algorithm maps each application page (bidder) to a SPM or Main Memory page (object).

Auction Algorithm

To illustrate capabilities of the auction scheme to solve the memory mapping problem, an auction algorithm with central management is presented in this section. The runtime system maps incoming applications to unutilized cores. Incoming applications send a list of their pages, alongside with the number of accesses, to the manager. The manager is responsible for creating objects and bidders –based on the above-mentioned model– and also for performing the auction algorithm. The basic auction algorithm, proposed by Bertsekas [15], is used to perform the auctioning. The auction algorithm used to determine SPM mapping is illustrated in Algorithm 6. In the beginning of this process, the manager creates objects and bidders of

the auction process based on number of accesses to each application page and the distance between application pages and SPM pages (Lines 1-6). In an iterative process, all unassigned bidders bid on the objects that maximize their benefits (Lines 7-16).

Assume memory page j (called object j from this point on) has price p_j and application page i (called bidder i from this point on) will be benefited a_{ij} by obtaining object j. So the net benefit of obtaining object j for bidder i is $a_{ij} - p_j$ and every bidder tries to get the maximum benefit by receiving object max_i (Line 11):

$$a_{imax_i} - p_{max_i} = \max_{j=1..n}\{a_{ij} - p_j\} \tag{3.1}$$

By receiving max_i, bidder i is happy and the system is in equilibrium if everybody is happy. The following process which finds the equilibrium for the entire system is called *auction*

ALGORITHM 6: Auction algorithm to determine SPM mapping

Input: all SPM_pages with initial price, all application_pages with number of accesses, placement of SPM_pages and application_pages on the platform;
Output: application_page to SPM_page mapping

```
1  objects = new_set();
2  bidders = new_set();
3  for sp in SPM_pages do
4      objects.add(create_object(sp));
5  for ap in application_pages do
6      bidders.add(create_bidder(sp));
7  while unassigned_bidder do
8      bids = new_set();
9      for i in bidders do
10         if unassigned(i) then
11             a_{imax_i} - p_{max_i} = max_{j=1..n}{a_ij - p_j};
12             max_i = arg max_{j=1..n}{a_ij - p_j};
13             v_i = a_{imax_i} - p_{max_i};
14             w_i = max_{j≠max_i}{a_ij - p_j};
15             p_{max_i} = p_{max_i} + (v_i - w_i);
16             bids.add(create_bid(max_i, p_{max_i}));
17     evaluate(bids);
```

algorithm.

The auction algorithm is an iterative routine. At the beginning of each iteration, if all bidders are happy, the auction algorithm will be terminated. Otherwise, all unhappy bidders try to maximize their benefit by bidding on their desired objects. Some unhappy bidder, for example bidder i, finds the best object max_i which maximizes the benefit (Line 12):

$$max_i = \underset{j=1..n}{arg\,max}\{a_{ij} - p_j\} \tag{3.2}$$

By obtaining max_i, the net benefit of bidder i will be (Line 13):

$$v_i = a_{imax_i} - p_{max_i} \tag{3.3}$$

Bidder i also finds the net benefit of obtaining second best object (Line 14):

$$w_i = \underset{j \neq max_i}{max}\{a_{ij} - p_j\} \tag{3.4}$$

Bidder i acquires object max_i –takes it from the previous owner– and increases the price of object max_i such that the net benefit of obtaining max_i becomes equal to obtaining the second best objects. The new price of object max_i will be (Line 15):

$$p_{max_i} = p_{max_i} + (v_i - w_i) \tag{3.5}$$

This can be viewed as an auction process in which bidder i bids on the object max_i with the value of $v_i - w_i$. The manager performs this process for all unhappy bidders.

If multiple bidders are equally interested in multiple objects, there might be a situation in which bidders exchange the objects without increasing the price of those objects. In that case, the auction process never terminates. To fix this problem, Bertsekas [15] proposed

a variation of auction algorithm inspired by real auctions. In this variation of auction algorithm, bidder i is almost happy if the net benefit of assigned object is within ϵ of its maximal net benefit:

$$a_{imax_i} - p_{max_i} \geq \underset{j=1..n}{max}\{a_{ij} - p_j\} - \epsilon \tag{3.6}$$

The auction process will terminate when all agents are almost happy. To incorporate this change, each bid value should be at least ϵ, in the other words, each bidder should raise the price of the desired object by $v_i - w_i + \epsilon$:

$$p_{max_i} = p_{max_i} + (v_i - w_i + \epsilon) \tag{3.7}$$

By termination of this process, each application page (bidder) is mapped to an object (memory pages) and the SPM mapping is complete.

Auction Modeling Challenges

Although auction algorithms are proven to be very efficient in resource assignment problems, some factors can affect the efficiency of this model significantly.

1. Defining Benefit Function:

   The auction algorithm tries to maximize the benefit for all bidders. Each bidder (application page) should define a benefit value for obtaining an object (SPM or main memory page) which should determine the relative preference of application pages for SPM pages. SPMs are more beneficial for application pages with higher number of accesses, so those application pages should have higher benefit values. Applications prefer to obtain closer SPM pages because the access latency is lower (ideally, everyone tries to get its local SPM). Therefore, closer SPMs should have higher benefit for

applications.

Considering the above mentioned characteristics of benefit values, we propose and use the following benefit function for every pair of bidder-object:

$$Benefit(bidder, object) = f(\#access, distance) = (-1) \times \#access \times distance \quad (3.8)$$

In this equation, *#access* is the number of accesses to the application page represented by the bidder and *distance* is the distance between the application page's home core and the SPM represented by the object.

2. Long Execution time:

   Applications might be consist of a large number of pages which translates to excessive number of bidders (and objects) in the auction mechanism. Higher number of bidders and objects results in more number of iterations to settle the assignments and prices. The increase in platform size exacerbates this problem: the potential number of concurrently running applications increases; as a result, a highly utilized many-core platform cannot obtain the SPM mapping in a timely manner.

   We use two techniques to reduce the execution time of the auction mechanism:

   (a) Increasing the minimum bid: the prices of objects become expensive with a higher rate and therefore other bidders become less interested in already-mapped objects. This persuades bidders (application pages) not to insist on a desired object (SPM Page) for a long time. Although this solution degrades the efficiency of SPM mapping, it can reduce the number of iterations significantly.

   (b) Grouping similar objects into a set: it reduces the number of objects significantly. Before starting the auction iterations, all SPM pages of a core have same price and benefit for applications. So we can group them in an object set. Each object set has only one price – representing the price of all objects in that set. Bidders

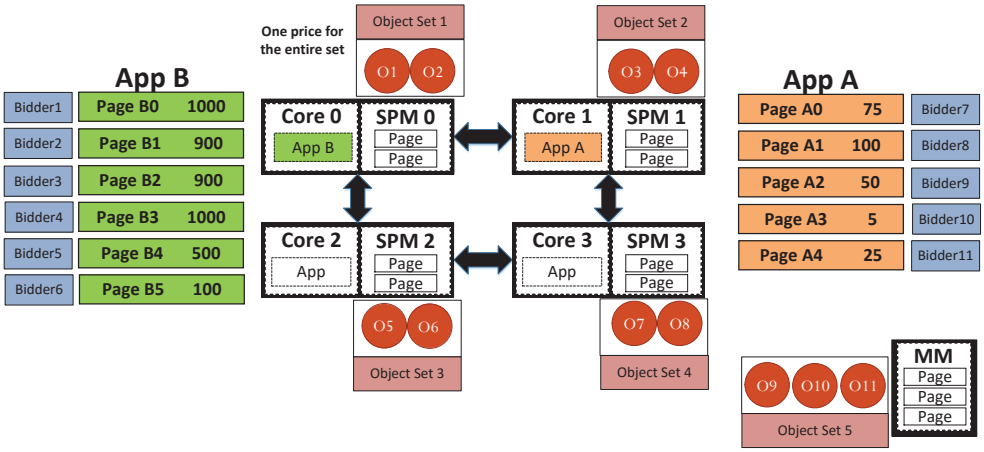


Figure 3.2: Using object sets to reduce number of objects

should bid on sets instead of objects. Figure 3.2 depicts this model.

The efficacy of auction-based model in solving SPM mapping problem is shown in Section 3.5.2.

3.4 Distributed Management of SPMPool

When platform size approaches hundreds of cores, a single central manager becomes incapable of managing the entire system with a reasonable overhead due to increased communication distance, network congestion, overhead of keeping data for every application, computation, etc. Therefore, using central management for memory mapping degrades system performance significantly and makes any management system unscalable. Auction mechanism as a distributed management system can be used to overcome this problem.

In this section we propose a two-layer distributed management scheme which divides the entire platform into multiple regions (Figure 3.3). The *Global Manager* assigns an arriving application to a region. Each region has its own *Local Manager* and pool of SPMs. The local manager is responsible for task and memory mapping inside its region. We first explore a

basic scheme where applications only can access SPMs inside their region (Section 3.4.1). In Section 3.4.2 we propose an auction-based management scheme to manage SPMPool memory mapping. Each region creates some bidders and is capable of bidding and obtaining SPM resources in other regions. This auction mechanism is managed in a distributed manner. The efficiency of the proposed management schemes are quantified in Section 3.5.

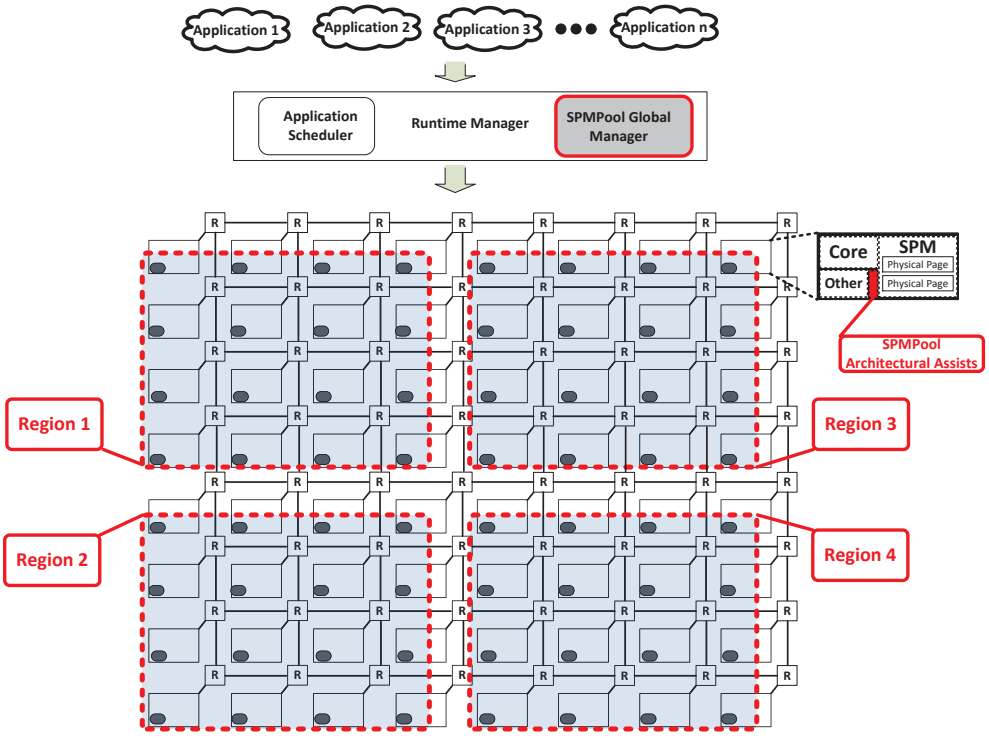


Figure 3.3: System view of distributed management

3.4.1 Non-communicative Distributed Pool Management

Figure 3.3 shows the system view of multi-region management for SPMPool. In this scenario, a pool of SPMs is composed of the aggregate on-chip SPMs of all cores in a defined region. Any executing application's memory can only be assigned to an SPM in the same region as the executing core; in the other words, no application can borrow on-chip memory from other regions. The regions that compose the pools of SPMs are determined at design time and fixed throughout the execution. In this section we demonstrate the two-level management

of SPMPool.

Global Manager

The Global manager is responsible for Mapping tasks to regions. The manager maintains a list of all application-region mappings and updates it upon stopping or starting of each application. Here, the global manager, as part of the runtime system, assigns the incoming applications to the region with the least number of applications. The advantage of this scheme is its simplicity but it does not consider the memory requirement of each application. The Global manager notifies the selected local manager about the incoming application. Likewise, when an application leaves the system, local manager signals the general manager.

Local Managers

Local managers are responsible for mapping the incoming tasks to cores within their region and also virtual to physical memory mapping of all applications assigned to the region. The local manager in this new scheme acts as a central manager for a single pool, introduced in Section 2.4.1. There is no need for local managers to communicate with each other, but they should communicate with the Global Manager. We implemented the described scheme for a 16x16 platform (details in Section 3.5.1). This platform is divided into four 8x8 regions. As a result, one global manager and four local managers are needed to supervise the SPMPool memory mapping. In Section 3.5.3, results of distributed management are compared to central management in terms of performance and overhead. The results show more than 75% reduction in memory migration overhead in some cases. Fixed size pools reduce the memory mapping overhead, but borrowing SPM resources from other regions can reduce memory access latency when memory resources of a pool are heavily used. Sharing memory resources between different pools cannot be managed with a central manager. In Section

3.4.2 a distributed approach for SPM sharing between different regions is proposed.

3.4.2 Auction-based Distributed Pool Management

In Section 3.4.1 a multi-pool scheme is introduced to manage SPMPool when the platform size reaches hundreds of cores. This two-layer management scheme is more efficient than central management (Section 3.5.3), but it does not exercise the full capacity of the platform. For example, if the applications mapped to pool A require significantly more memory resources than applications mapped to the neighbor pool B, applications of pool A will suffer from poor memory mapping. The ability to share SPM resources between different regions can potentially increase the memory mapping performance. To avoid the drawbacks of a central management, a distributed scheme should be exploited to improve the memory mapping.

The capability of running the auction mechanism in a distributed manner, makes it a good candidate for SPMPool management. Figure 3.4 shows the system view of distributed multi-region management for SPMPool. The basics of this management scheme can be described as follows:

- The entire platform is divided into multiple regions. These regions are fixed; the cores forming a region are determined at design time. Each region has a *local manager*.
- One *global manager* and multiple *local managers* (one manager per region) create a two layer management scheme. The global manager assigns an arriving application to a region. The local manager is responsible for task and memory mapping inside its region. Local managers are connected through a virtual mesh network.
- The aggregate of on-chip SPMs assigned to the applications running in a region, composes the pool for that region. SPMs located inside a region are divided into two

groups: exclusive SPMs and shared SPMs; exclusive SPMs are only used by the applications inside the region while shared SPMs can be used by any application in local or neighbor regions.

- To avoid starvation and provide a minimal performance for each region, half of the SPMs inside a region are exclusive and half of them can be shared with another region. In this work, while we used half of all SPMs for exclusive access, of course other fractions could be used and investigated as well. All shared SPMs of each region is considered as one group and can be assigned to only one neighbor region. For example, assume region A is in vicinity of regions B and C; all shared SPMs of region A can be used locally –by applications inside region A– or can be used by applications in region B or region C exclusively, and not by all A, B, and C at the same time.

- Each local manager of a region is the corresponding agent for all applications and SPMs inside that region. Each agent generates one object representative of all shared SPMs inside its region and creates bidders on behalf of applications inside its region. These bidders can bid on the local object and also on the remote objects – represented by neighbor agents. A distributed auction mechanism assigns objects (shared SPMs) to bidders (regions).

In the following section a distributed auction mechanism is proposed to reshape pools of SPMs based on memory requirement of each region.

Agent Negotiation Using Distributed Auction

All local managers, which we call *agents* henceforth, need to generate the SPM mapping for the applications running in their region. The set of SPMs assigned to each region (pool of SPMs) should be determined before generating any SPM mapping algorithm. In this work,

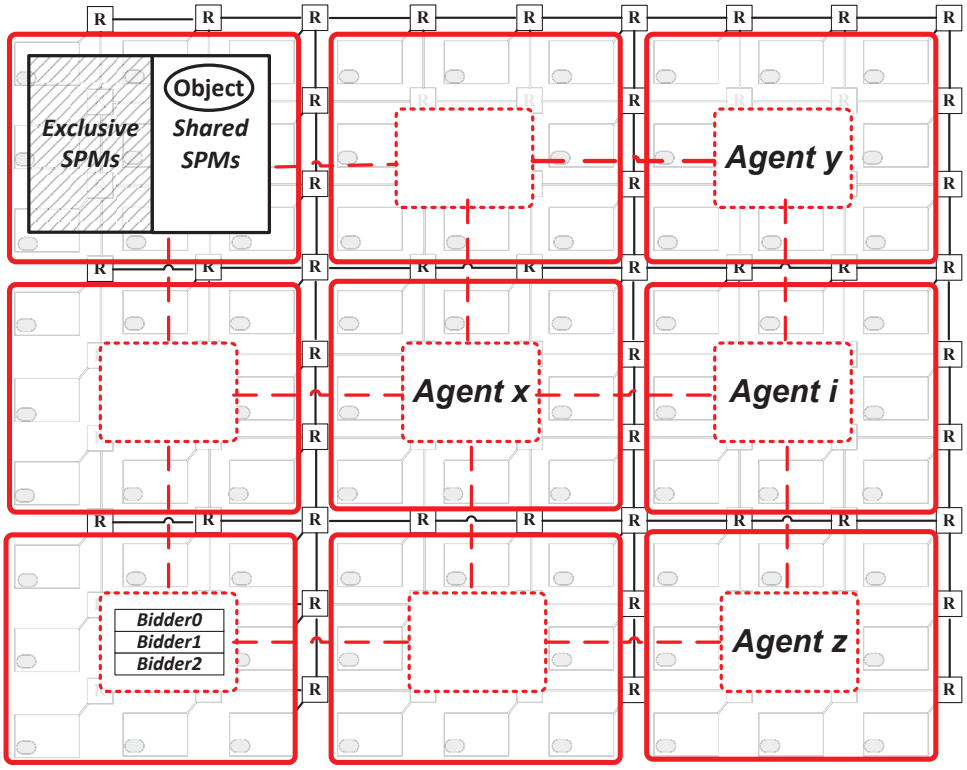


Figure 3.4: System view of distributed pool management. The platform is divided into different regions. Each region has a local manager which behaves as an agent. Half of the SPMs in each region can be shared. They form a single object. Agents can bid on objects to obtain the SPM resources in other regions. The auction mechanism is managed in a distributed manner.

we adopt a basic distributed auction mechanism, introduced by Zavlanos [132], and propose a mechanism to alter SPM pools based on variable requirements of each region.

In the system illustrated by Figure 3.4, we assume all agents are connected through a mesh network with diameter d (d=4 here). If the memory usage of one pool changes significantly (by entering, exiting, or phase change of applications), the corresponding agent starts a new auction process. The common routine in agent i can be described as Algorithm 7.

At setup time, agent i creates objects O_{i0} (Lines 4-5, representative of shared SPMs in region i) and O_{i1}, O_{i2}, and O_{i3} (Lines 6-8, representative of offchip memory which helps to terminate the auction process). Agent i should keep the price and highest bidder for all

objects in every neighbor agent (x, y, and z).

We allow each agent to obtain the shared SPMs (objects) of local and neighbor regions. In other words, each agent can obtain shared SPMs of $\#neighbors + 1$ regions (4 for agent i). Therefore, agents need to create bidders for maximum attainable objects (4 bidders for agent i) and specify a benefit function for them. Each bidder represents a set of application memory pages: to create a bidder, we sort the application pages inside the region based on

ALGORITHM 7: Distributed Auction algorithm executed in Agent i

Input: all objects, their prices (p), and the maximum bidder (b) for each object: from local agent and all neighbors agents; all applications and their pages in region i; assignment of local bidders (α)

Output: object to bidder mapping

```
1  if setup then
2      local_objects = new_set();
3      local_bidders = new_set();
4      O_i0 ← Object(initial_SPM_price);
5      local_objects.add(O_i0);
6      for j = 1..3 do
7          O_ij ← Object(initial_Offchip_price);
8          local_objects.add(O_ij);
9      app_pages ⇐ concatenate_{applications_in_region_i} {application_pages};
10     sorted_pages = sorted(app_pages);
11     for j= 0.. 3 do
12         b_ij ← Bidder(sorted_pages[k(i+1)..k(i+2)−1]);
13         local_bidders.add(b_ij);
14     setup = false;
15 for j in objects do
16     p_j(t+1) = max_{received_from_all_paths} {p_j(t)};
17     index = pathmax_{received_from_all_paths} {p_j(t)};
18     b_j(t+1) = path_index.b_j(t);
19 for ix in local_bidders do
20     if b_{α_ix(t)}(t+1) ≠ ix then
21         α_ix(t+1) = arg max_{j=1..n} {a_(ix)j − p_j(t+1)};
22         b_{α_ix(t+1)}(t+1) = ix;
23         p_{α_ix(t+1)}(t+1) = p_{α_ix(t+1)}(t) + (v_ix − w_ix + ε);
24 Send_assignments_to_all_neighbors();
```

the number of accesses to them and choose a block of those pages. Every block of pages corresponds to a bidder. If region i contains k exclusive SPM pages and k shared SPM pages, we can presume that the top k pages go to the private SPMs, so we can define the bidders as in Equation 3.9 (Lines 9-13):

$$\begin{aligned} app\_pages \Leftarrow \underset{applications\_in\_region\_i}{concatenate} \{application\_pages\} \\ sorted\_pages = sorted(app\_pages) \\ b_{i0} \leftarrow Bidder(sorted\_pages[k..2k-1]) \\ b_{i1} \leftarrow Bidder(sorted\_pages[2k..3k-1]) \\ b_{i2} \leftarrow Bidder(sorted\_pages[3k..4k-1]) \\ b_{i3} \leftarrow Bidder(sorted\_pages[4k..5k-1]) \end{aligned} \tag{3.9}$$

In Equation 3.9, the "Bidder" function creates a bidder using the input pages. The benefit of obtaining an object is a function of distance and number of access to that object. For example, if object O_{xp} is representative of shared SPMs in region x, the benefit of receiving O_{xp} by bidder b_{i0} is: $f(\sum_{j=k}^{2k-1} sorted\_pages[j], distance(i,x))$.

Assume in agent i, $\{O_{x0}, O_{x1}, ..., O_{i0}, O_{i1}, ..., O_{z3}\}$ are the objects in local and neighbor agents. At the beginning of the t_{th} iteration, $p_j(t) \geq 0$ is the price of object j, $b_j(t)$ is the highest bidder for object j, and $\alpha_{ix}(t)$ are the assigned objects to all bidders created by agent i. All agents have their own local view of objects and bidders; upon updating any price, each agent sends its local updates to its neighbors. After receiving new updates, agent i at time t executes this routine:

1. Agent i updates prices and highest bidders for all objects (Lines 15-18). Agent i might get prices from different paths, the highest received price will be the price for each

object:

$$p_j(t+1) = \max_{received\_from\_all\_paths} \{p_j(t)\} \tag{3.10}$$

$$index = \underset{received\_from\_all\_paths}{pathmax} \{p_j(t)\} \tag{3.11}$$

$$b_j(t+1) = path\_index.b_j(t) \tag{3.12}$$

2. if $b_{\alpha_{ix}(t)}(t+1) == ix$, the assignment has not been changed, so $\alpha_{ix}(t+1) = \alpha_{ix}(t)$ and this iteration stops here. Otherwise, we go to the next step.

3. if $b_{\alpha_{ix}(t)}(t+1) \neq ix$, it means that bidder ix is no longer assigned to $\alpha_{ix}(t)$ and it should find the object which maximizes its net benefit (Line 21):

$$\alpha_{ix}(t+1) = \underset{j=1..n}{arg\,max}\{a_{(ix)j} - p_j(t+1)\} \tag{3.13}$$

4. Set $b_{\alpha_{ix}(t+1)}(t+1) = ix$ and increase the price for object $\alpha_{ix}(t+1)$ (Lines 22-23):

$$p_{\alpha_{ix}(t+1)}(t+1) = p_{\alpha_{ix}(t+1)}(t) + (v_{ix} - w_{ix} + \epsilon) \tag{3.14}$$

where v_{ix} and w_{ix} are obtained by equations (3.3) and (3.4).

Agent i sends the new prices to its neighbors. Every agent performs this process locally and the process continues until reaching stable prices for all objects. If objects' prices do not change for d iterations (d is the diameter of the network), prices have been stable and the process can be stopped. If b_{i0}, b_{i1}, b_{i2}, and b_{i3} are the bidders created by agent i, the collection of objects assigned to these bidders determines the pool of SPMs for agent i. The memory mapping for region i can be changed after the termination of this process. With this

algorithm, the auction mechanism can be implemented without any global synchronization and shared memory.

3.5 Experimental Setup and Results

3.5.1 Experimental Setup

The SPMPool simulation infrastructure (described in Section 2.5) is updated to include auction-based memory mapping (Section 3.3) as one of the memory mapping policies in SPMPool in addition to existing policies. The SPMPool simulator is changed to support multi-pool memory management as described in Section 3.4. To support multi-pool management, the Global Manager initially creates regions, associated agents, and a virtual mesh network of these agents. The Global Manager also assigns incoming applications to the region with the least number of executing tasks. Each agent administrates the SPM mapping for all applications inside its local region and also negotiates with other agents to borrow additional SPM resources. All information related to the auction mechanism (objects, bidders, etc.) is stored in each agent. A buffer in each agent keeps the latest assignment data (objects/prices/bidders) received from all neighbor agents. A software routine in each agent executes the distributed auction algorithm (Algorithm 7).

For our experiments, we used a variety of benchmarks from the MiBench suite [55] to execute on an ARM architecture configuration and combined them to create diverse workloads. The selected benchmarks represent different embedded application domains that have varying memory-use characteristics. To generate each workload, benchmarks and their associated entry time were selected randomly, all running to completion. Table 3.1 lists the benchmarks and number of different data sets in the experiments. Memory traces were extracted by executing the benchmarks with the gem5 simulator [19].

Table 3.1: List of benchmarks used from MiBench suite

Application Type	Benchmark	Data Sets	Application Type	Benchmark	Data Sets
Consumer	jpeg	2	Network	dijkstra	2
Consumer	typeset	2	Network	patricia	2
Consumer	lame	2	Telecomm	fft	2
Automotive	qsort	2	Telecomm	gsm	2
Automotive	bitcount	2	Security	blowfish	2
Automotive	susan	2	Office	stringsearch	2

3.5.2 Central Auction-Based Memory Mapping

To evaluate the efficiency of auction-based memory mapping, introduced in Section 3.3, we compared it to the other heuristics previously used in SPMPool memory mapping (Section 2.4.1). Figures 3.5 and 3.6 illustrate the comparison of memory access latency between most-accessed, simplified most-accessed, and auction-based mapping policies in different utilization points for 4×4 and 8×8 platforms respectively. These results are normalized to the simplified most-accessed policy.

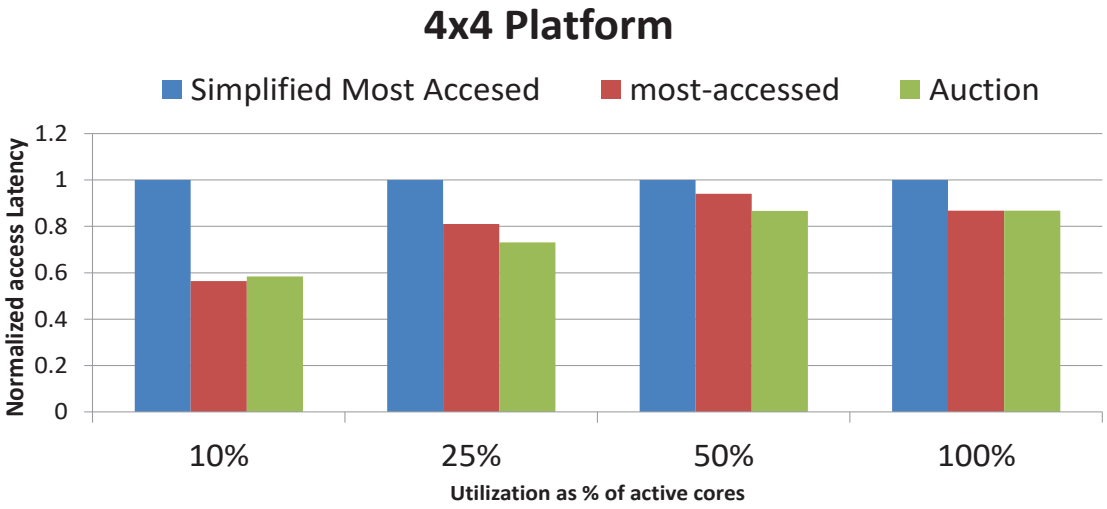


Figure 3.5: Comparison of total memory access latency of different mapping policies, normalized to the simplified most-accessed policy for 4x4 platform

At very low and very high utilization points (10% and 100%), auction-based policy does not show a significant improvement. But at medium utilization points, auction-based policy

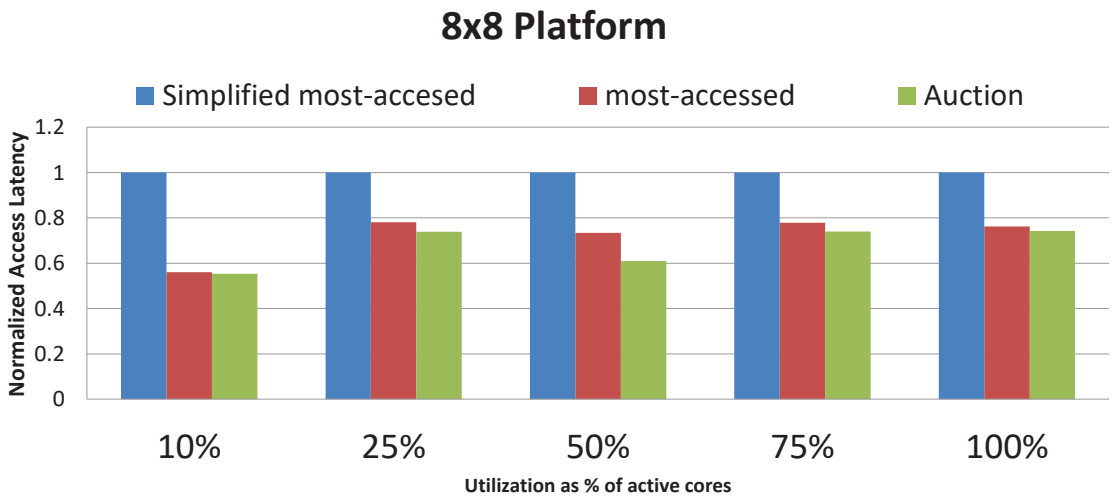


Figure 3.6: Comparison of total memory access latency of different mapping policies, normalized to the simplified most-accessed policy for 8x8 platform

outperforms other policies by up to 12%. Although auction-based memory mapping improves the memory access latency, for larger platform sizes, the execution time of this method becomes intolerable.

3.5.3 Distributed Multi-Pool Management

We implemented the distributed management schemes as described in Section 3.4 for 16x16 platform using both non-communicative and auction-based schemes. The entire platform is divided into four regions. The Global manager maps each task to the region with the minimum number of running tasks. Each region has a Local manager and tasks are mapped to a randomly selected core within each region. Other experimental setup is similar to the setup described in Section 3.5.1.

Figure 3.7 illustrates the overall memory access latency of Non-Communicative (NC) and Auction-based distributed management in different utilization points. In the low utilization point (10%), the auction-based approach slightly improves the total memory access latency, but in the medium utilization points (25% to 75%), the auction-based approach outperforms

the non-communicative approach by up to 24%. Moving toward 100% utilization, none of these approaches can improve the performance significantly.

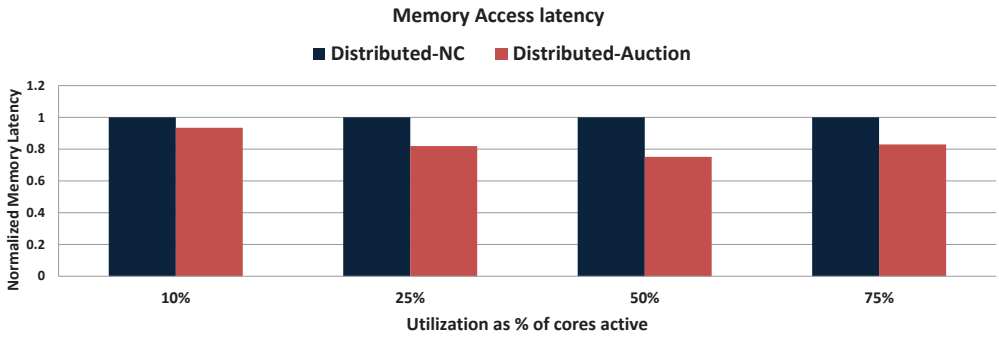


Figure 3.7: Memory access latency, using different distributed management schemes

To investigate the sensitivity of auction-based memory management to the region size, we first divided the entire platform to four 8×8 regions. We repeated the experiment for sixteen 4×4 regions. Figure 3.8 depicts the memory access latency of auction-based distributed management schemes for different region sizes. In all utilization points, higher number of regions result in a better memory access latency because of more optimization opportunities. Higher number of regions also increases the communication overhead of the management scheme which is studied in Section 3.5.4.

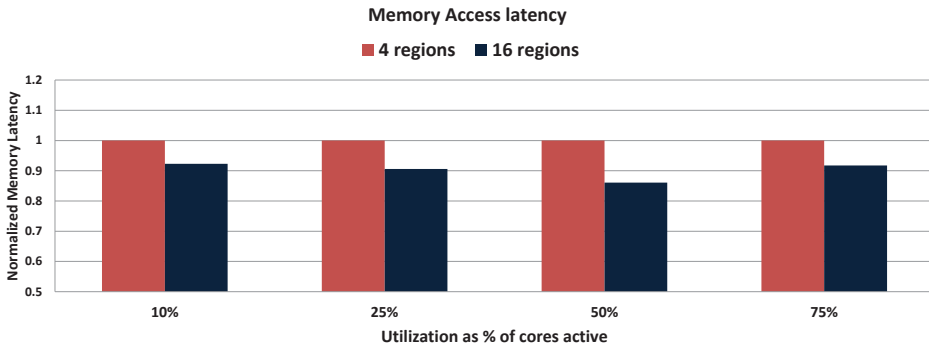


Figure 3.8: Comparison of memory access latency in auction-based distributed management for different region sizes

3.5.4 Overhead

Communication, memory migration, and storage are some sources of overhead imposed by SPMPool memory management schemes. Figure 3.9 compares the communication overhead of Non-Communicative (NC) and Auction-based distributed management schemes. This overhead accounts for the number of packets sent only for management purposes, between applications and agents or between neighbor agents. It is assumed that each packet is 256 bytes. The auction-based scheme improves the memory access latency significantly and has a negligible communication overhead compared to the non-communicative scheme.

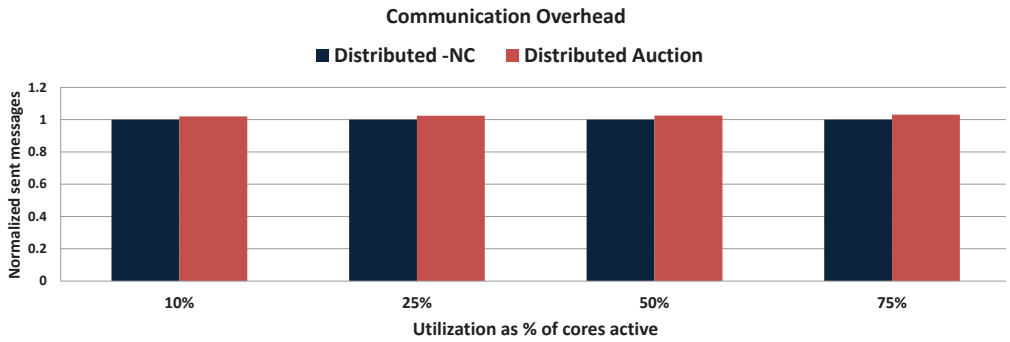


Figure 3.9: Communication overhead of distributed management, measured as the number of packets sent for management purposes between application-agent or agent-agent. Auction-based distributed scheme endures up to 3% more overhead over non-communicative distributed scheme

A comparison of communication overhead for different region sizes in auction-based distributed management of 16×16 platform is shown in Figure 3.10. In 100% utilization point, communication overhead of 16 regions is 10% higher than the communication overhead of 4 regions.

After any change in memory mapping, memory migration is inevitable. Figure 3.11 shows the total number of pages transferred between off-chip and on-chip memory. Memory migration overhead of distributed management is far less than that of central management. When utilization increases, memory migration overhead jumps with a higher rate in distributed management because the mapping becomes more similar to the central management scheme.

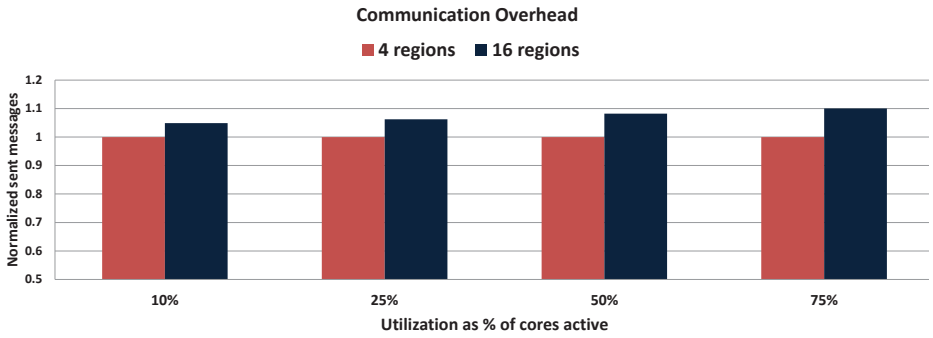


Figure 3.10: Comparison of communication overhead for different region sizes in auction-based distributed management. Communication overhead is measured as the number of packets sent for management purposes.

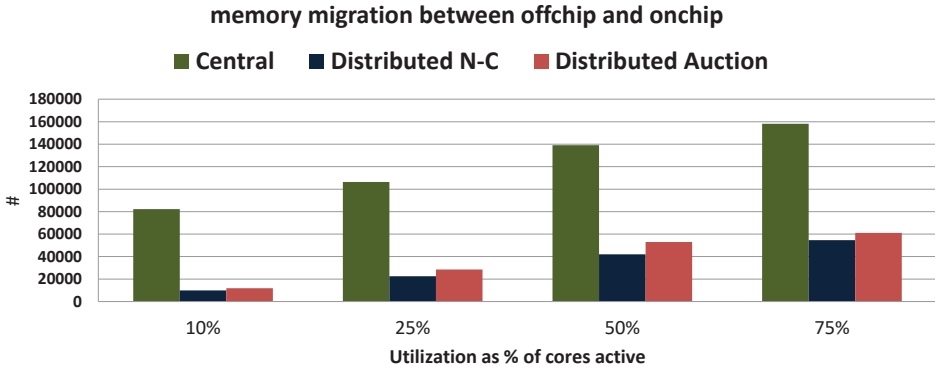


Figure 3.11: Memory migration overhead – Number of pages transferred between off and on chip

In non-communicative distributed management, the storage overhead of memory mapping policies increases linearly with the number of pools. To implement most accessed policy in this work, storage overhead of non-communicative distributed management is 75% less than that of central management (Figure 3.12). To conclude, our initial investigations show that the distributed management makes SPMPool approach scalable since it imposes far less overhead than the central management scheme.

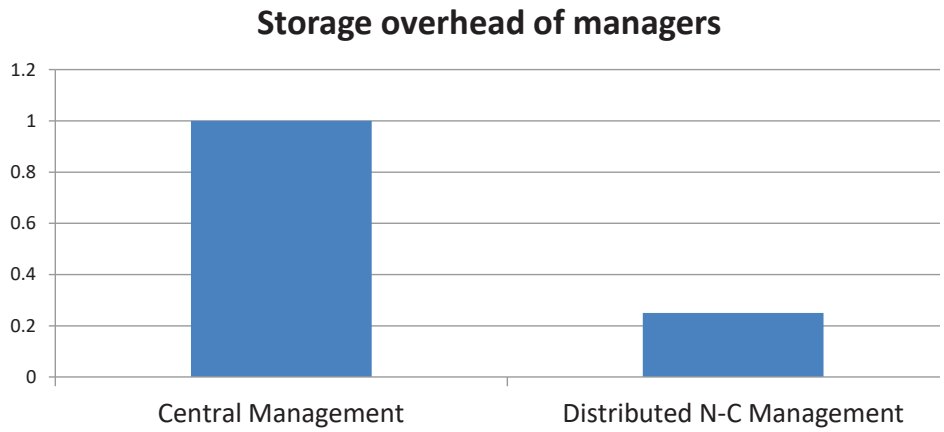


Figure 3.12: Storage overhead comparison of central and non-communicative distributed management

3.6 Conclusion

In this chapter, we proposed different auction-based schemes for memory management in many-cores systems. After showing the advantages of auction-based algorithms in solving memory mapping problem, we proposed a distributed management scheme based on auction algorithm. In this scheme, the entire platform is divided into different regions and a distributed auction mechanism is used to share the SPM resources between different regions. Our preliminary experiments on a 16×16 platform executing different benchmarks from Mibench suite show that auction-based distributed management achieves up to 24% improvement in memory access latency over non-communicative distributed management with a negligible increase in communication overhead. The distributed schemes proposed in this chapter provide a scalable solution for memory management of many-core platforms by mitigating the overheads of central management scheme.

For experiments in this chapter, we assumed that half of the SPM resources in each region are exclusive to that region. This assumption helps to introduce the concept of auction-based distributed management. A sensitivity analysis of the fraction of exclusive SPMs and also the placement of shared SPMs are subjects of ongoing research.

Chapter 4

Memory Phasic Behavior

4.1 Introduction

High processing capability is a crucial requirements for modern systems. Technology scaling and low power design have enabled modern devices to deploy an ever-increasing amount of hardware resources integrated on a single chip, in the form of multi- and many-core platforms. These platforms are expected to support workloads consisting of numerous concurrently executing applications with varying resource requirements. Memory is a critical resource, and is often the bottleneck for applications. To efficiently manage and effectively allocate limited memory resources both within and between applications, we should be able to extract the memory usage information of all applications.

Memory requirements may vary highly between different multimedia applications. When sharing memory resources, the ability to prioritize each individual memory page –accessed by running applications– enables us to assign memory resources to each application proportional to the memory requirement of that application. Moreover, it can guide us to the efficient memory mapping in the presence of Non Uniform Memory Access (NUMA).

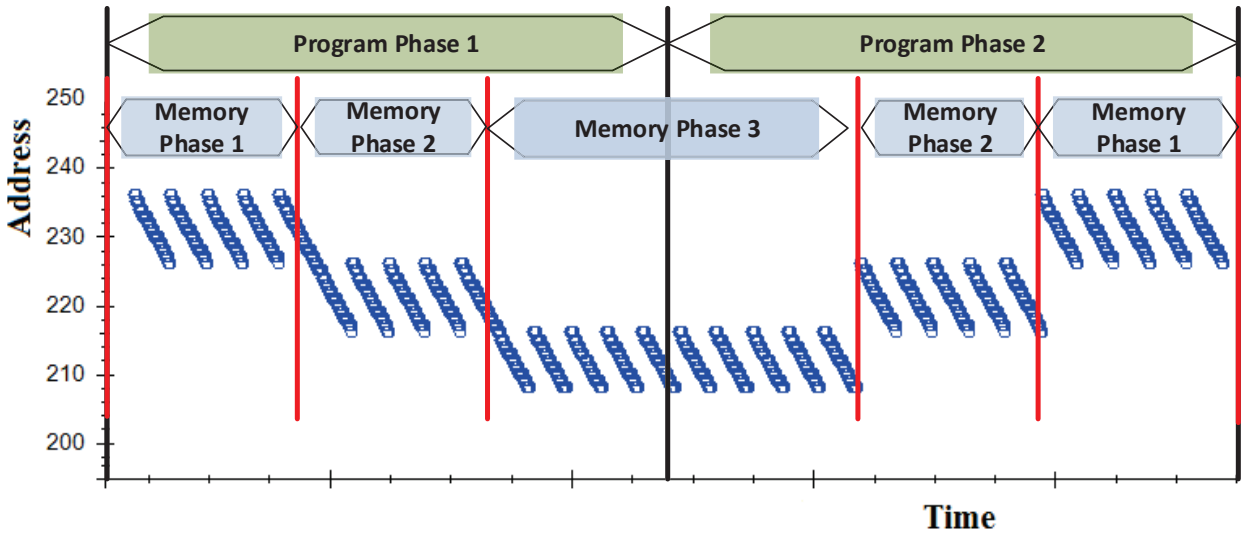


Figure 4.1: Memory Phases and Program Phases for the same code snippet (Multimedia Source Code 8).

Applications have very dynamic behavior and the memory requirements of each individual application varies over time throughout the course of execution. For instance, rendering high-motion scenes needs more memory resources than rendering still scenes. Most applications can be divided into different phases. Repetitive, or *phasic*, behavior of applications has been the subject of research for several years. Typically the phasic behavior is investigated with respect to the structure of applications (i.e., application's basic blocks and control flow) or execution working set. Although these efforts have proven useful, detecting patterns in the execution of an application by tracking instructions (or Basic Blocks) does not always capture the memory phases of the application. Typical program phasic behavior extraction techniques do not differentiate between program and memory behavior [9, 40, 110], and thus miss opportunities for more aggressive memory management in the face of high memory demands.

For example, consider Figure 4.1 which compares the memory and program phases extracted from a sample routine (Multimedia Source Code 8) that accesses a multidimensional array. This routine represents array computations typically present in many multimedia ap-

plications. In this example, the memory usage of the application is not aligned with the instruction-based program phases – program phases don't represent changes in memory usage. Using such program phases may lead to poor memory management. Therefore, there exists a need for detection of *memory phasic behavior* based on the memory usage.

Previous research efforts have been made to detect application phases based on memory accesses. These efforts include capturing cache miss rate, detecting reuse distance, etc. Although these works inherently try to find memory phases, they're unable to predict and prioritize individual pages in the working set of applications.

In this work we present an online memory-phase detection scheme with the ability to prioritize the memory working sets of concurrently running applications [120]. The rest of this chapter is organized as follows: Section 4.2 will go over the research efforts on phasic behavior of applications and the motivation for detecting and using memory phases. Section 4.3 defines memory phases and proposes an offline method to capture memory phases of applications. Section 4.4 will present an online method to detect and classify memory phases at runtime. A use case of memory phases in SPMPool is presented in Section 4.5. Experimental results of memory phasic behavior is discussed in Section 4.6. Section 4.7 ends this chapter with some concluding notes.

4.2 Related Work and Motivation

The repetitive nature of program behavior has been investigated for several years. Program phase detection techniques are used for diverse purposes including power reduction, simulation, micro-architectural adaptation, etc. [70, 56, 112, 106].

Some Program-Instruction-Based methods have been proposed for detecting phases. These methods monitor control flow and executed instructions of a program in intervals. Sherwood

et al. [110] uses Basic Block Vectors (BBVs) to find out if there is a considerable change in the executing basic blocks of a program during an interval. Lau et al. [80, 81] enhance this method. Instead of capturing all basic block transitions, Ratanaworabhan et al. [101] propose to track only Critical Basic Block Transitions. Dhodapkar et al. [38] define a program phase by the set of instructions observed in an interval (labeled as instruction working set). Cho et al. [27] use wavelet-based analysis to differentiate between phases. Balasubramonian et al. [9] use conditional branches as a metric to detect program phase changes.

In spite of these efforts, tracking instructions (or basic blocks) does not always capture the memory phases of applications. A single instruction in an application[1] commonly accesses different data throughout the course of execution. For example, if the inputs change prior to entering a memory intensive block, no program phase change will be detected while the application may experience a significant change in memory working set. Alternatively, the executing instructions of an application might change while working on the same set of data. An example of this behavior is shown in Multimedia Source Code 8 which is typical of kernel computations in multimedia applications.

Multimedia Source Code 8: Example

```
 1:  for (int k =0; k < 3; k ++){
 2:    for (int j = 0; j < 5; j ++){
 3:      for (int i =0; i < 10000; i=i+5){
 4:        sum += arr1[k][i +j];
 5:      }
 6:    }
 7:  }
 8:  for (int k =2; k >=0 ; k --){
 9:    for (int j = 0; j < 5; j ++){
10:      for (int i =0; i < 10000; i=i+5){
11:        sum += arr1[k][i +j];
12:      }
13:  }
```

If we divide this code into phases by its basic blocks, each inner *for* loop (Lines 3-5, 10-

[1] In this work, we define an application as a single thread assigned to a core, and use the terms application and program interchangeably.

12) would make up a single phase. However, the outer *for* blocks (Lines 1-2, 8-9) impact the index of the array being accessed (Lines 4, 11). This means that all accesses to each respective array occur within a single program phase. This concept is illustrated in Figure 4.1. Memory phases capture changes in the memory working set of the program much more accurately than program phases in this case.

Some phase change detection techniques are based on observed performance statistics of applications (e.g., cache miss/hit rate, IPC, etc) [9, 40]. These techniques can be used for optimizations such as cache reconfiguration, but are highly sensitive to architectural parameters of the platform (e.g., cache size).

Phase-driven optimizations have also been used in many cache tuning schemes. While most of these schemes rely on instruction-based phase detection techniques, some try to monitor memory related characteristics of the program. Shen et al. [109] and Huffmire et al. [65] monitor memory accesses and use the memory reuse distance for phase detection. Although these techniques have some indication of memory phases, they are offline profiling methods that analyze sample executions. In addition, they do not capture the memory working set of an application and do not provide sufficient information to prioritize different applications in the context of shared memory in multi-cores.

Compile-time static analysis and offline profiling are not sufficient for determining the memory usage patterns of an application. Also, the source code of applications is not always readily available. For instance, smartphone users download many applications every day without having any analysis of those applications. Additionally, the input dependence of memory accesses and dynamic allocation necessitates runtime adaptation even in the presence of prior knowledge about the application. Therefore a useful memory phase detection scheme must observe and extract memory behavior at runtime.

This chapter presents a runtime memory phase detection technique that discovers similar-

ities and changes in the memory requirements of an application over time during execution and provides information to improve the efficiency of dynamic memory mapping without relying on any offline profiling, static analysis, or other additional information. We provide an importance measure for different memory pages that indicates the relative utilization of each page both within and between applications.

4.2.1 Contributions

- We define memory phases and propose offline and online methods for detection and prediction of memory phases in multimedia systems.
- We present a scheme for inter-application prioritization of data memory using the extracted approximate working set information.
- We illustrate the efficacy of memory-phase driven memory mapping in managing single- and multi-core platforms equipped with Software Programmable Memories (SPMs). Preliminary experiments show the potential efficacy of exploiting memory phasic behavior by up to 45% improvement in memory access latency of executing Mibench benchmarks on these platforms.

4.3 Memory Phases

Due to the time-varying memory requirements of an application, using the metrics that represent average behavior of that application can be inefficient in any memory-related optimization. For example, tracking the total number of accesses to each memory page is not a good representative of access patterns, because access locality changes over time. Our goal is to find periods of execution within an application that have a similar memory working set,

whether these periods are consecutive or not. In this section we define the notion of memory phases and propose an offline method to detect memory phases of an application.

4.3.1 Memory Phase Definition

The working set of an application is the collection of memory pages being accessed during a certain period of time. To be precise, the working set of an application at time t will be the set of memory pages accessed during the period of $[t-\tau, t]$, for a predefined time period τ. A memory phase is the period of time in which the working set of an application within that period is similar (obviously, without defining *similarity*, this definition is not complete).

In the working set, not all pages are utilized equally. Assume that during the period of $[t_1-\tau, t_1]$, page X has been accessed n_1 times and page Y has been accessed m_1 times and $n_1 \gg m_1$. Also, during the period of $[t_2-\tau, t_2]$, page X has been accessed n_2 times and page Y has been accessed m_2 times and $n_2 \ll m_2$. In this case, although the working sets consist of the same pages, these working sets are not similar.

To incorporate the number of accesses to each page when considering similarity, we define the Weighted Working Set (WWS) at time t as in Equation 4.1.

$$WWS[t] = \{(page, \#access) j for pages accessed in [t-\tau, t]\}. \tag{4.1}$$

From now on, WWS is used to define memory phases instead of working set.

We define a memory phase as a period of time in which the weighted working set throughout the period is similar – with a predefined similarity metric. In other words, each application has a WWS at each point in time during its execution and we want to cluster those WWSs into similarity groups. With this definition, the behavior of an application from a memory access pattern point of view is nearly uniform during a memory phase. Any memory

phase change marks a significant change in the application's memory access pattern.

4.3.2 Offline Memory Phase Detection

To detect memory phases offline, a clustering technique suitable for the WWS data should be used. In the offline method, all the data accessed throughout the course of execution can be obtained by profiling and there is no time or storage constraints. Our offline approach to the clustering problem is summarized in the following steps:

1. **Memory profiling:** In the first step, the set of all memory accesses of an application is obtained by profiling. Using the memory accesses, WWS for each point in time can be calculated (throughout the entire course of execution).

2. **Reducing data size:** There exists a WWS for each point in time. Based on the value of τ, a single WWS may contain many accesses. To reduce the execution time and required memory, the WWS data should be reduced. There are two ways to reduce WWS data: 1) In any WWS, pages with the most accesses are the main representatives of that working set. To reduce data, we only maintain a list of k pages with the highest number of accesses. After this reduction, each WWS consists of information for only k pages. 2) There is no need to compute WWS for every point in time. We can compute the WWS every m seconds. This reduces the required storage by a factor of m. In the following steps, all calculations will be on the reduced data.

3. **Creating a vector representation of each WWS:** If during the course of execution, total number of P distinct pages are held in all weighted working sets, each WWS can be represented by a vector with P dimensions. Within the WWS, if page p is accessed n times, the value of p_{th} dimension in the representative vector will be n.

4. **Clustering WWS vectors and determining phases:** In this step, similar WWS vectors are clustered in one group. Therefore, it is necessary to quantify the similarity between two vectors. All vectors consist of positive numbers (number of accesses), therefore we can use the dot product of vectors to find the angle between two vectors:

$$A.B = |A| \times |B| \times cos\theta \Rightarrow$$

$$cos\theta = \frac{A.B}{|A| \times |B|} \Rightarrow$$

$$cos\theta = \frac{\sum_{i=1}^{P} a_i b_i}{|A| \times |B|}$$

   $cos\theta$ is the similarity metric for our clustering mechanism. If $cos\theta = 1$, the angle between two vectors is zero and they have the maximum similarity. If $cos\theta = 0$, the angle between two vectors is 90 degrees and they have the minimum similarity.

   Using this similarity metric, we execute a hierarchical clustering algorithm (HCA) [53] on WWS vectors. As opposed to k-means clustering [89], this method does not require the number of clusters.

   Each WWS group represents a phase: if WWS of two periods of time are in the same group, those periods belong to the same phase.

The efficiency of this offline memory phase detection technique is presented in Section 4.6.3.

4.4 Online Detection of Memory Phases

The offline phase detection method requires a considerable amount of memory and execution time and therefore is not feasible to incorporate in runtime software. More importantly, due

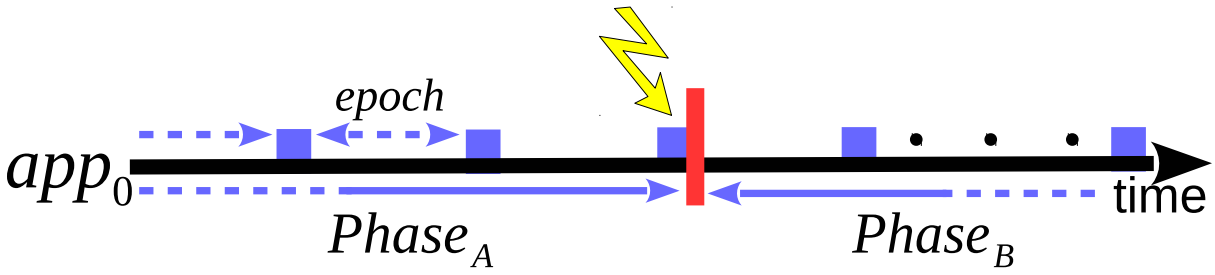


Figure 4.2: Timeline of a single application with memory phase detection. Memory accesses are monitored throughout each epoch, and at the end of each epoch we determine if a phase change has occurred.

to the input dependence of memory accesses, the complete memory profile of an application is not available prior to execution, creating the need for an online method. In this section, we present a simplified online adaptation of the offline method with reduced overhead to detect and use memory phases at runtime.

4.4.1 Memory Phase Detection Scheme

Figure 4.2 illustrates the timeline of the light-weight runtime phase detection scheme in relation to an executing application. Memory accesses are captured throughout fixed observation periods, or *epochs* (details in Section 4.4.1), and a phase-change-detection routine runs at the end of each epoch (details in Section 4.4.1). Phases span multiple epochs.

To reduce overhead, the only memory access information maintained for phase change detection is an approximation of the weighted working set of the most accessed pages for both the current epoch and the current phase. At the end of each epoch, the intersection of these two lists is used to determine whether a phase change has occurred. If no phase change is detected, this continues for the next epoch. If a phase change is detected, the phase signature is computed and stored. The weighted working set lists are reset for capturing memory accesses in the next epoch.

In the online approach, capturing memory accesses (Section 4.4.1) is an approximation

for the first three steps of the offline approach. The captured memory accesses are used to detect the phase change and find the similarity to the previous phases (Sections 4.4.1 and 4.4.1) which is an approximation for the clustering step (step 4) in the offline approach.

Capturing Memory Accesses

In our runtime phase detection technique, we identify phases by the memory locations that are accessed the most. Therefore we must identify the memory pages with the highest number of accesses in each phase. This is accomplished by maintaining weighted working sets consisting of the most accessed pages in each epoch.

Finding the k pages with the highest number of accesses during a period of time is analogous to finding the k most frequently occurring numbers in a series of numbers. If n different numbers appear in a series of numbers during an observation period, it can be proven that finding the k most frequent numbers requires $O(n)$ storage [32]. In this instance, n represents the number of unique pages accessed during an epoch, which can be unmanageably large. Therefore, finding the exact k pages with the highest number of accesses is impossible at runtime.

Although finding the exact solution is infeasible due to storage overhead, approximate solutions exist that can find effective solutions with limited storage. In this work we use an approximate solution to find the k most frequently occurring numbers in a series of numbers and adapt it to find the k pages with the most number of accesses during an epoch. In our case, the approximate solution is formulated in Algorithm 9.

Existing algorithms solve variations of this approximate problem [32, 25]. The Frequent Algorithm [32] tries to find all items in a sequence whose frequency exceeds a *1/k* fraction of the total counts. We propose an algorithm to capture the most accessed pages during an epoch by adapting the Frequent Algorithm (Algorithm 10). In this algorithm, k counters

ALGORITHM 9: Capturing memory accesses – approximate solution

Inputs :

Given a stream of n accesses a_1, a_2, ..., a_n
Frequency of an access is $f_i = |\{j|a_j = i\}|$

Output:

The accesses occurring with a frequency $\geq \phi$ percent of all accesses.
The exact $\phi - frequent = \{i|f_i > \phi n\}$
The approximate $\phi - frequent = \{i|f_i > (\phi - \epsilon)n\}$

are paired with k unique memory addresses. Upon each memory access, the address of the access is compared to all saved addresses. If a match is found, its corresponding counter will be incremented (Lines 3-9). If the new access is not matched with any of the existing addresses and all k pairs are not yet occupied, a pair will be initialized with counter value 1 (Lines 11-15). If all k pairs are allocated to distinct memory addresses, all the counters are decremented by 1. In this case, if one of the counters becomes zero, the corresponding pair will be initialized with the new address and counter value of 1 (Lines 17-24). Memory access monitoring and profiling can be implemented in either software or hardware. Figure 4.3 shows the hardware implementation of our Frequent Algorithm adaptation. Compared to software implementation, this implementation imposes some area overhead, but increases the performance of this routine significantly. Using the adapted Frequent Algorithm, it is possible to approximate the k most accessed pages with only k storage elements and k counters. As expected, in some cases, this approximate algorithm cannot produce the exact result: a few of the detected pages might not be among the actual k most accessed pages. In Section 4.6.4 we show that the accuracy of this approximate algorithm is comparable to the precise solution. Therefore this approximate algorithm can be used to find the top k most accessed pages, using limited storage.

ALGORITHM 10: Frequent Algorithm

```
Input:
  accesses -- sequence of memory accesses
Output:
  Page[] -- k pages with most frequent accesses
  Counter[] -- k counter values associated with
               each page
 0:  access = new_memory_access
 1:  while (access){
 2:    Found = False;
 3:    for (int j = 0; j < k; j++){
 4:      if (access == Page[j]){
 5:        Counter[j]++;
 6:        Found == True;
 7:        Break;
 8:      }
 9:    }
10:    if (!Found){
11:      empty = FindEmptySpace();
12:      if (empty != -1){
13:        Page[empty] = access;
14:        Counter[empty] = 1;
15:      }
16:      else{
17:        for (int j = 0; j < k; j++){
18:          Counter[j] --;
19:          if (!Found && Counter[j] == 0){
20:            Page[j] = access;
21:            Counter[j] = 1;
22:            Found = True;
23:          }
24:        }
25:      }
26:    }
27:    access = new_memory_access
28:  }
```

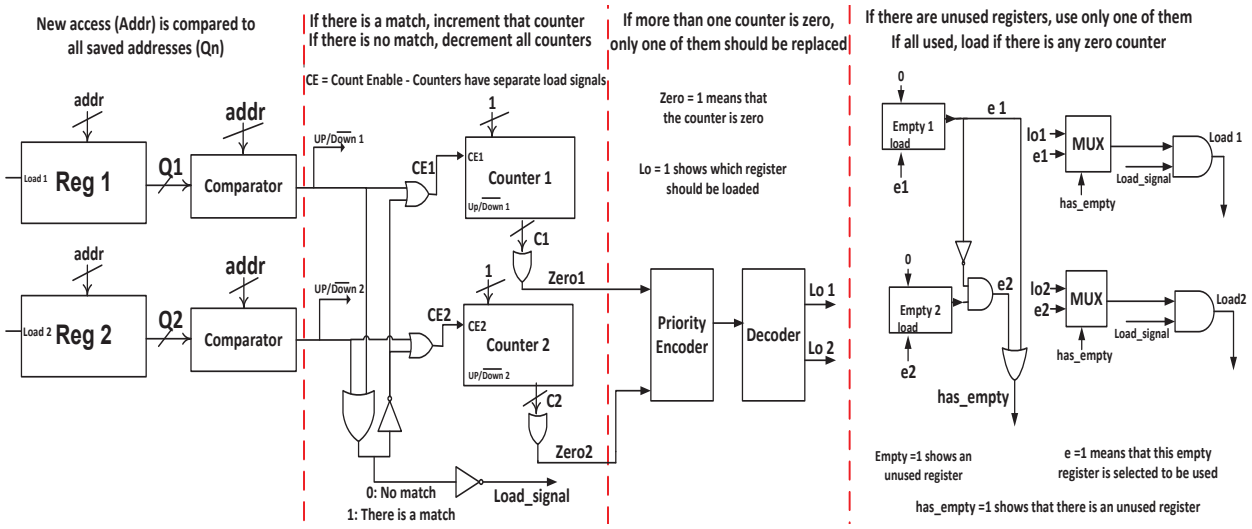


Figure 4.3: Hardware implementation of capturing memory accesses for working set size of 2. To expand the working set, Counter/Reg 2 should be replicated.

Phase Change Detection

At the end of each epoch, we use the captured memory access information to detect whether or not a phase change has occurred. Due to memory and time restrictions at runtime, we cannot realistically employ the clustering technique used in the offline method. Therefore we make the following simplifications:

- To compute the similarity between two Weighted Working Set vectors, we use the Manhattan distance instead of dot product.
- The WWS vector for each epoch is only compared with the current phase's WWS — we must maintain the WWS of the current phase.

This routine is outlined in Algorithm 11 and is sensitive to the number of counters k, threshold τ, and epoch length. In this work we only consider stable phases — a phase change is not triggered unless the length of the new phase is more than one epoch. This reduces overhead, especially for the case when an application does not have any stable phase. The threshold to trigger a phase change can be application specific. The programmer/compiler can set this

threshold explicitly, or it can be adaptive. In this work, we used a fixed threshold –found empirically– for each application. As the phase change detection scheme is triggered at the end of each epoch, a short epoch can impose a considerable overhead and should be avoided.

ALGORITHM 11: Phase Change Detection

Inputs :

Previous epoch WWS: Pages with the most accesses in the previous epoch, $A_1, ..., A_k$, and the number of accesses to those pages $n_1, ..., n_k$

Normalized current phase WWS: Pages with the most accesses in the current phase, $X_1, ..., X_k$, and the number of accesses to those pages $m_1, ..., m_k$ divided by number of epochs in the current phase.

Output:

if

$$\frac{\sum_{A_i \neq X_j}(n_i \quad or \quad m_i) + \sum_{A_i = X_j} |n_i - m_j|}{Epoch \quad Length} > \tau$$

then

$\Rightarrow Phase\ Change = True$

Weighted Working Set Prediction

For most memory optimization methods, simply detecting a phase change is not sufficient — we also must be able to predict the weighted working set of an application in the new phase. One solution is to choose the first epoch of the new phase as the representative of that phase. However, this might not be representative of the entire phase as phases span multiple epochs.

If a memory phase change is detected, two possibilities for the new phase exist: 1) it is a completely new phase, and 2) the application has experienced this memory phase in the past and it is returning to that memory phase again. This can be used for a better prediction of WWS in the new phase.

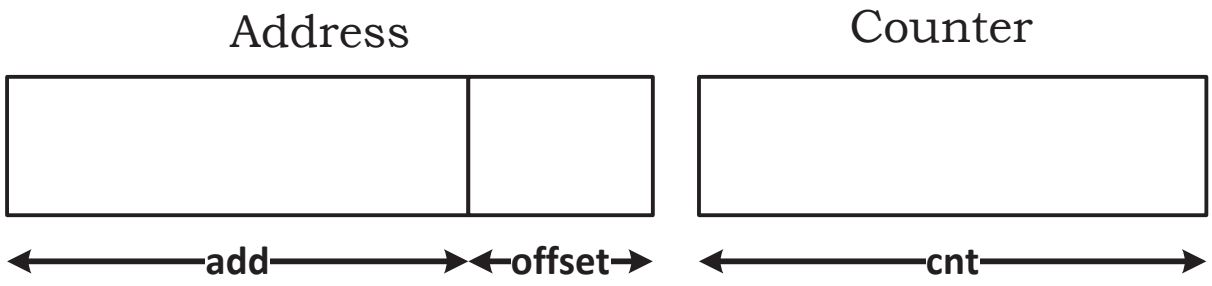


Figure 4.4: Tuple of `(add, cnt)` is the representative of each page in WWS.

In this work, we exploit a history-based prediction scheme to anticipate the WWS of the new phase. The intuition can be described as follows: if the application is experiencing a new phase for the first time, the weighted working set in the preceding epoch is a sufficient estimate of the weighted working set in the new phase. On the other hand, if we are returning to a phase, we use historic information to find the WWS of the phase from its previous execution.

To implement the history-based prediction scheme, we need to store the WWS of previous phases. Each page in the weighted working set is stored as a 2-tuple containing page address and weight: `(add, cnt)` in Figure 4.4. The signature of weighted working set of a phase is then defined by all 2-tuples of k pages in the WWS.

To distinguish between a returning phase and a completely new phase, the WWS signature of the previous epoch is compared with the stored signatures of previous phases. If the Manhattan distance between signatures of the previous epoch and one of the phases is within a threshold, a returning phase is detected and the stored WWS for that phase is used as the predicted WWS of the current phase. Otherwise, the phase is considered as a completely new phase.

4.4.2 Overhead of Online Phase Detection

Storage overhead

Storage overhead is incurred by our online technique for storing k pages of WWS for both the current phase and epoch. A page entry in WWS can be stored as a `(add, cnt)` pair. Assuming 64-bit addresses and 4KB pages, $64 - 12 = 52$ bits are required to store a page address. Using 20 bit counters, we need $20 + 52 = 72$ bits (or 9 bytes) to keep any `(add, cnt)` pair. With $k = 16$, $9 \times 16 = 144$ bytes are required to store all information of a WWS. This storage requirement can be reduced by using a smaller k value. In the history-based prediction scheme, the maximum number of *max-h* weighted working sets are stored for comparison with the current phase (*max-h* $= 5$ in this work). Total number of *max-h* $\times k \times 9$ bytes are required to implement this scheme. This is reasonable – in Section 4.2 we show that small values of k and *max-h* can be effective.

Computation Overhead

Computation overhead is incurred at runtime by capturing memory accesses and the phase change detection routine. Using the hardware depicted in Figure 4.3, capturing memory accesses are feasible in few cycles. Computation overhead of phase change detection routine is proportional to k^2 – where k is the number of pages in WWS. For predicting the weighted working set upon a phase change, the current phase should be compared to *max-h* stored phases and the required time is proportional to $k^2 \times$ *max-h*. Both k and *max-h* are very small numbers. Furthermore, none of these routines are in the critical path of the application. Therefore the computational overhead of our phase detection scheme is not significant.

4.5 Memory Phase driven SPM Mapping: a Use Case

We now illustrate the opportunity to exploit memory phases through a use case for software controlled memory mapping. Multicore platforms enable concurrent running of different applications in emerging mobile devices like smart-phones and tablets. These devices can execute multimedia and non-multimedia applications simultaneously. With the ability to extract the phasic memory behavior of applications at runtime, we can exploit the information to effectively assign application data to memory resources for unpredictable workloads. Memory phase detection allows us to capture the temporal variation in memory requirements of different applications. It can be used to handle the dynamic nature of multimedia applications. We discuss this challenge in the context of multi-core systems containing distributed on-chip Software Programmable Memories (SPMs).[2]

Figure 4.5 illustrates the system level view. Each core has SPM as on-chip memory and contains necessary components to enable sharing and utilization of all on-chip SPM space. Moreover, cores are equipped with the memory phase detection scheme. Note that the Runtime Manager has a Memory-Phase-Aware SPM manager that maps application data to the SPMs in the underlying tiled multi-core hardware platform.

In complex workloads, applications can start and stop at any time. The Runtime Manager is responsible for application mapping and scheduling, as well as SPM mapping: deciding what application data should reside on chip and what resides in main memory. An SPM mapping is determined based on memory phase information provided by each core. The weighted working set of each memory phase can be obtained and used to prioritize application pages at runtime. The SPM mapping routine is performed periodically when applications start or stop, and possibly when they change memory phase – updating the SPM mapping for every phase change of every application is excessive. The goal of our SPM mapping

[2]The optimization approaches discussed can apply to any NUMA management.

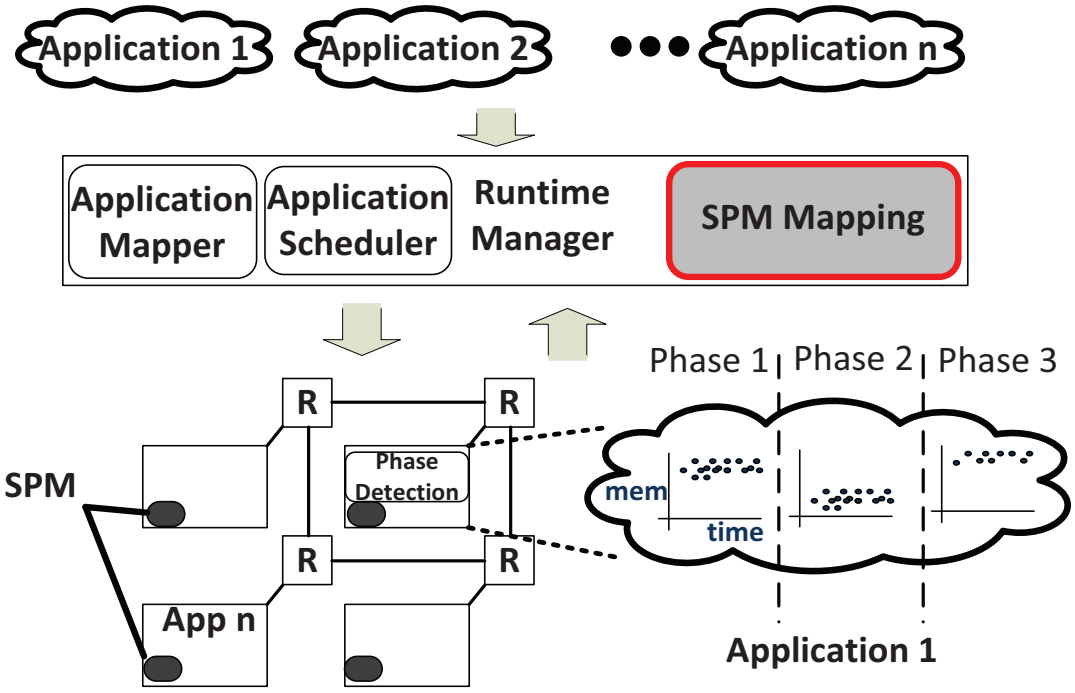


Figure 4.5: System level view of memory phase driven SPM mapping in multi-core systems.

scheme is to reduce the average memory access latency of the entire workload, which in turn can reduce energy consumption due to reduction in off-chip memory accesses.

The phase change detection routine executes on each active core, and the Runtime Manager receives messages indicating phase changes. These messages include the estimated working set of the new memory phase. The benefits of an updated mapping is estimated by the Runtime Manager, and if the potential performance improvement exceeds a threshold, the SPM mapping will be changed.

4.5.1 Compute New SPM Mapping

Each application maintains a weighted working set of its current memory phase. The weight of each page in WWS, which is the estimated number of accesses to that page, can be used to prioritize different pages of applications. When triggered, the Runtime Manager executes the mapping algorithm.

The mapping algorithm maps application pages to on-chip SPM slots with the goal of achieving the lowest overall memory access latency. In Section 2.4.1, we showed that this optimization problem is NP-hard and finding optimal solution for this problem is infeasible at runtime, therefore we developed a heuristic for the memory mapping policy.

The simplified description of this policy –called *Most-Accessed Mapping policy*– is specified in Algorithm 12. This mapping policy compares WWS of all executing applications and keeps pages with the highest weights on chip (Lines 1-6). The *pages* array contains all the pages in the executing applications (Lines 1-3). After sorting these pages based on their weights (Line 4), the first N pages with the highest weights will be assigned to *onchip_list* (Line 5) and the rest will be assigned to *offchip_list* (Line 6).

Most-Accessed Mapping policy tries to map those pages in SPMs as close as possible to their home core (Lines 8-14). For each *page* in the *onchip_list*, if the *page*'s home core has an empty slot, it will be mapped to the home core (Lines 7-10). Otherwise, the closest core with an empty SPM slot will be found (Line 12) and the *page* will be mapped to that core (Line 13). All the pages in *offchip_list* will be mapped to off-chip memory.

The proposed scheme is a first attempt at solving the problem. In a better implementation of this scheme, overheads of re-mapping should be estimated. If the benefits do not outweigh the overheads, a re-mapping should not be triggered. This modification is left for future work.

The memory phase detection scheme can extract the access pattern of applications. Therefore, benefits of this scheme is not limited to SPM mapping. For example, in a system with a data cache, if only a portion of a large data block is important, recurring accesses to that block –much larger than cache size– leads to a poor performance; the memory phase detection scheme can be used to extract the memory access pattern and improve the data caching performance.

ALGORITHM 12: Most-Accessed Mapping Policy

```
Input:
  WWS[] -- WWS of m executing applications
Output:
  SPM Mapping

 1:  pages = []
 2:  for (i = 0; i < m; i++)
 3:    pages = pages.append( WWS[i] )
 4:  sorted_page_list = sort(pages, key=page_weight)
 5:  onchip_list = sorted_page_list[0..N-1]
 6:  offchip_list = sorted_page_list [N..end]
 7:  for (page: onchip_list){
 8:    home = home_core(page)
 9:    if (home.has_empty_SPM_slot())
10:      home.assign(page)
11:    else{
12:      remote = home.closest_non_full_core()
13:      remote.assign(page)}
14:  }
```

4.6 Experimental Setup and Results

4.6.1 Experimental Setup

We developed a Java and Python-based simulation infrastructure to extract phases and simulate workloads at memory operation level. It uses a software implementation of Algorithm 10. The simulator is capable of extracting memory phases both off- and on-line, and program phases offline. The workloads are simulated given the memory and instruction traces of all applications. Our memory-phase simulation infrastructure is capable of evaluating overall memory access latency of user-specified workloads on a variety of configurations in multi-core platforms.

For our experiments, we used a variety of multimedia and non-multimedia benchmarks from the MiBench [55] suite that have varying memory-use characteristics. Table 4.1 lists the benchmarks and number of different data sets used when capturing traces. We also used

Table 4.1: List of benchmarks from MiBench suite [55]

	Benchmark	Number of data sets
Multimedia Applications	jpeg	2
	susan	2
	lame	2
	fft	2
	typeset	2
Non-Multimedia Applications	qsort	2
	dijkstra	2
	bitcount	2

a variety of synthetic benchmarks with different memory behavior, and combined them to create diverse workloads. A combination of these benchmarks are used to evaluate the phase driven SPM Mapping. To generate each workload, benchmarks and their associated entry time were selected randomly, all running to completion.

Memory and instruction traces are obtained by executing each benchmark using the gem5 simulator [19] for the x86 architecture. The memory traces serve as inputs to our simulator. The simulator has configurable parameters for the system under test — we evaluate platforms from 1 to 16 cores. For all of our experiments, we assume all virtual and physical memory is divided into 4KB pages, and each core has 16KB of local SPM. There are no data caches present. CACTI [92] was used to obtain memory bank access latency, and Noxim [47] was used to calculate on-chip communication latency.

4.6.2 Experimental Goals

We conducted experiments to 1) highlight the difference between program and memory phases and 2) illustrate the potential advantage of memory phases over program phases in multimedia applications. We also evaluate the effectiveness of our approximate online memory capture method.

4.6.3 Program Phase vs Memory Phase

As stated in Section 4.2, the method proposed by Sherwood et al. [110] is one of the most used Program-Instruction-Based methods, proposed to detect phasic behavior of applications. This method captures Basic Block Vectors (BBVs) of an application in some defined intervals. The size of BBVs is reduced by linear projection and k-means clustering is used to group similar intervals into a phase.

To illustrate the difference between memory phases and program phases, we implemented this program phase method and compared it with our memory phase detection scheme proposed in Section 4.3. To find the BBVs, the structure of the application must be known, in contrast with our method that does not require structure or instruction information. Also by using hierarchical clustering, we do not need to make an assumption about the number of clusters.

As an example, Figure 4.6 shows memory and program phases for a snapshot of the *qsort* benchmark. Every point in this plot represents a memory access. The dashed lines show the boundaries of program phase changes and solid lines show the boundaries of memory phase changes (blocks labeled with m1 to m5 show distinct memory phases of the application). While memory phases successfully capture the phasic behavior of this application, program phases are incapable of finding a meaningful change in the memory access pattern. Because memory phases are only detected when observation periods (epochs) end, memory phase boundaries may not be totally aligned with memory access pattern changes. In another example, Figure 4.1 illustrates memory and program phases for the code shown in Source Code 8. Similar to qsort, program phases do not correspond to the memory access pattern of the program.

To quantify the benefits of using memory phases, we execute four benchmarks individually on a single core with SPM as on-chip memory. We use both program and memory phases to

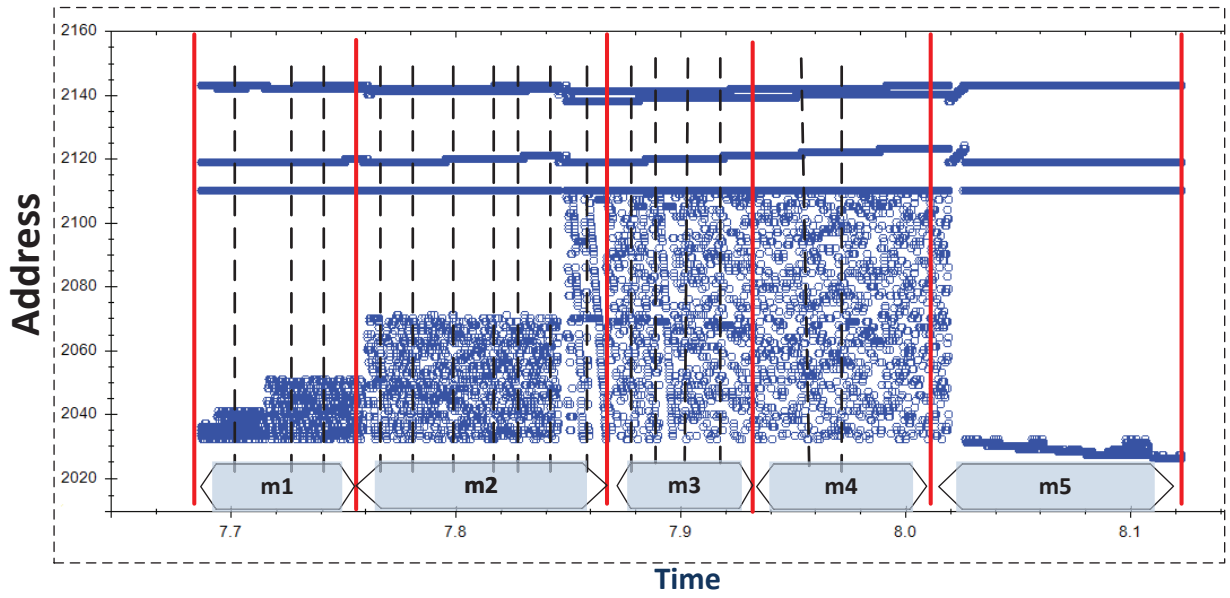


Figure 4.6: Memory Phases (red vertical bars) vs Program Phases (black dashed vertical bars) in qsort benchmark.

trigger allocation of the SPM, assuming the most accessed data is automatically allocated into the SPM every time it is triggered. Total memory access latency of these three different allocation triggers is compared in Figure 4.7:

- No phase detection (application has one phase)
- Memory phases
- Program phases

To maintain fairness, we tried to assign the parameters such that the number of distinct phases in both methods is similar. Note that the overhead of executing the phase change detection scheme is not considered in these results.

By using memory phases, up to 45% improvement in memory access latency is achieved, compared to the case without using any phasic information. The nature of memory phases is to detect drastic change in memory working set, which would indicate the most beneficial

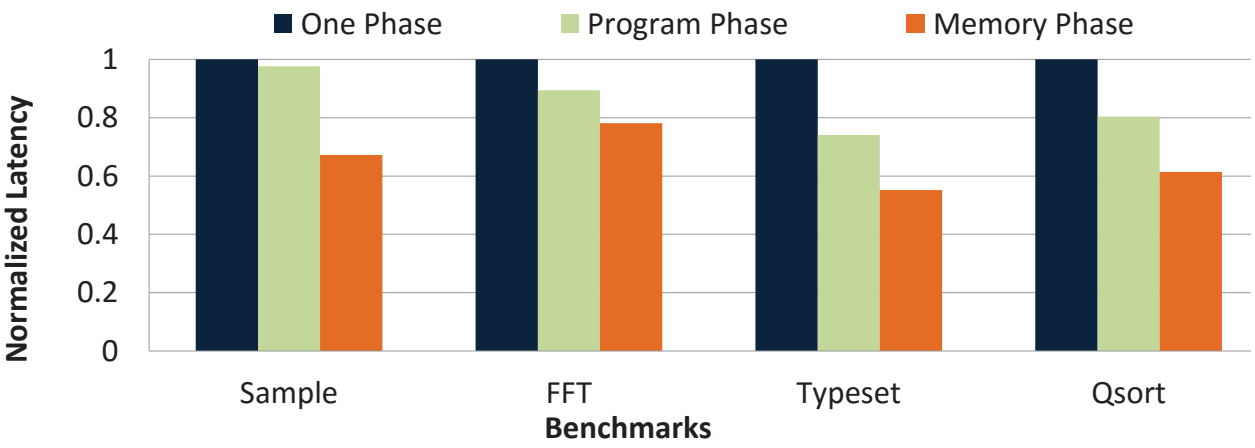


Figure 4.7: Memory access latency improvement using Program- and Memory-triggered SPM mapping normalized to static mapping.

time to update the contents of on-chip memory. More results can be found in our technical report [119].

4.6.4 Accuracy of Capturing Memory Accesses

The modified frequent algorithm –which we use when capturing memory accesses at runtime– is an approximate algorithm, and its accuracy in finding the most accessed memory pages significantly impacts the ability of online phase detection scheme. We compare the ability of this algorithm (implemented using limited storage) with the precise algorithm (implemented using unlimited storage) in finding $k = 8$ pages with the highest number of accesses throughout an epoch. At the end of each epoch, we compare the two algorithms' lists and keep track of matches and mismatches. The accumulation of these numbers throughout the execution of each benchmark provide us with an accuracy measure. Figure 4.8 shows the percentage of correct results generated by the modified frequent algorithm through the course of execution:

$$\%Correct\_Results = \frac{\#matches}{\#matches + \#mismatches} \tag{4.2}$$

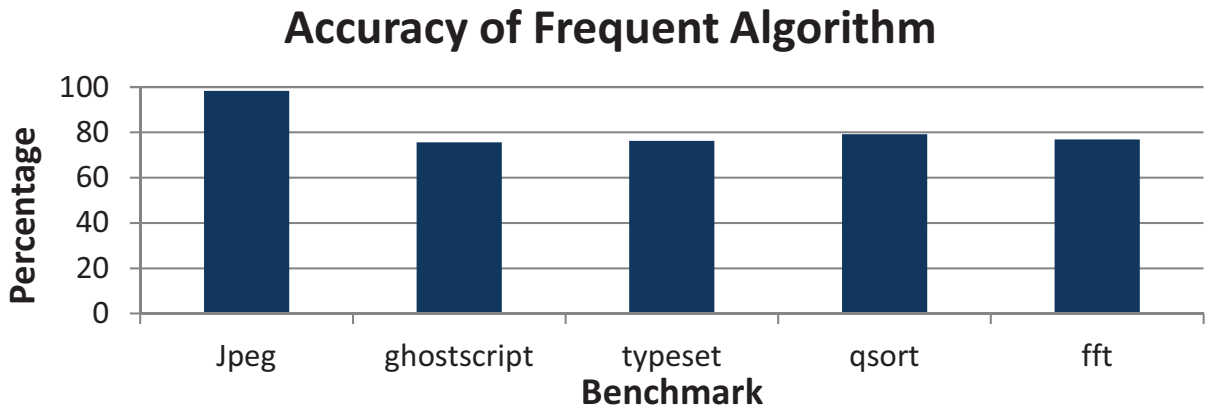


Figure 4.8: Percentage of correct answers generated by frequent algorithm in finding top-8 most accessed pages of all epochs throughout the course of execution.

In all cases we have more than 78% accuracy and in the *Jpeg* benchmark accuracy is near 100%. These results show the efficacy of the frequent algorithm in capturing the most-accessed pages.

4.6.5 Latency Reduction of phase driven SPM Mapping

In this section we show the efficacy of memory phase driven SPM mapping introduced in Section 4.5. Memory phase driven mapping is compared with this scenario: one phase for the entire execution of applications –not considering phasic behavior. Without a memory phase detection scheme, we have no information about memory usage of applications at runtime. So we ideally assume the weighted working set of applications are known.

Figure 4.9 shows the memory access latency of different utilization points for 4×4 multi-core platform. Different applications from Table 4.1 and synthetic benchmarks are combined to form the workload.

As these graphs indicate, memory phase driven SPM mapping has better results in all

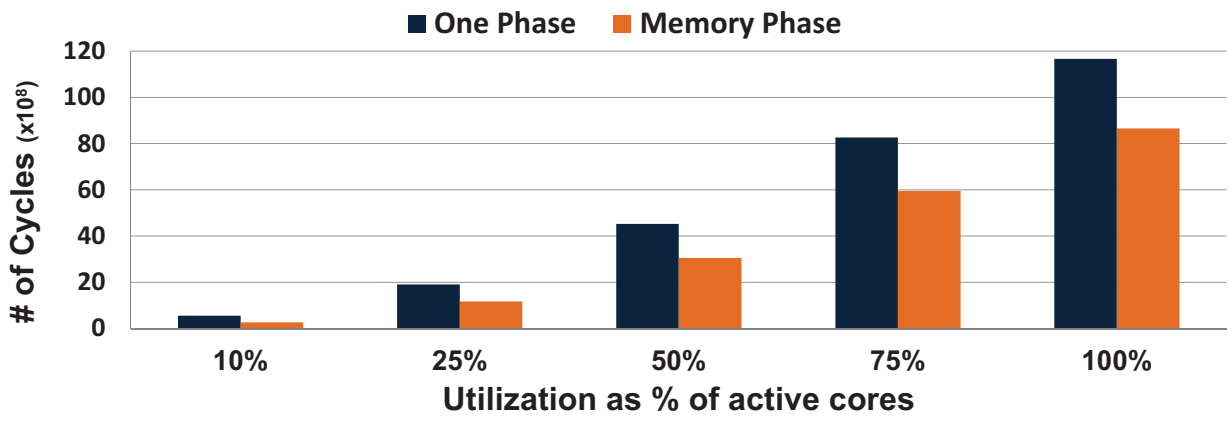


Figure 4.9: Memory access latency for different SPM mapping methods in 4×4 platforms.

cases (up to 50%), even though the non-phasic case is ideal and cannot practically be implemented at runtime. This shows us that the memory phase change detection and signatures successfully identify beneficial pages to place in SPM at appropriate times and WWS of applications –obtained by our memory phase detection scheme– can prioritize different pages efficiently. More results can be found in our technical report [119].

4.7 Conclusion

Many-core platforms allow concurrent execution of multiple applications that enter and exit at unpredictable times. Since each application has temporal variation in its memory usage and working set, these memory footprint variations need to be captured and exploited in order to allow for effective use of the limited on-chip memory resources. In this chapter, we first presented the notion of memory phases and distinguished them from program phases. We presented a light-weight online memory phase detection scheme that can be integrated into the runtime system to enable exploitation of memory phasic behavior. We then showed how this memory phase detection scheme can be used for effective sharing of distributed on-chip software controlled memories (SPMs) for multi-core platforms, by developing a memory-phase-driven SPM mapping scheme.

Our experiments on workloads with varying intra- and inter-application memory-intensity shows that using phase detection schemes can reduce memory access latency up to 45% for configurations up to 16 cores. This access latency reduction originates from accessing onchip memories instead of going offchip, which also results in the reduction in power and energy consumption.

Chapter 5

Concluding Notes and Future Directions

The number of transistors on a chip have increased steadily over the past decades. Smaller and faster transistors have led to higher power densities which cease the frequency increase of the processors. This frequency stall, along with hitting the power and ILP wall, put an end to single-core processors. it has been projected that in 10 years, there will be more than 30 CPU and 200 GPU cores in mobile devices, and more than 100 CPU cores in micro-servers. With these developments, memory subsystems in many-core platforms will face different challenges.

Keeping the coherency of shared data becomes very expensive when moving towards hundreds of cores, and coherence protocols can be the major barrier for scalability of many-core platforms. Central memory management schemes also become very inefficient which is due to increased communication distance, increased computation and storage, and network congestion, to name a few.

Generally, memory requirements for different applications vary significantly from one an-

other. In addition to this between application memory variation, each individual application over the course of execution, also exhibits temporal variation of its memory requirement.

Any effective memory management system on a many-core platform should assign available onchip memory resources to numerous concurrently executing applications based on their memory requirements. To address these issues, we introduced a comprehensive runtime solution for memory management of many-core systems.

5.1 Main Contributions

The use of Software Programmable Memories (SPMs) alleviates the need for coherence protocols in many-core systems. However, it introduces new challenges to the memory subsystem of many-cores. We proposed a runtime memory management of SPM resources in many-core platforms in the presence of unpredictable workloads. The key contributions of this work are listed as follows:

- SPMPool: A runtime memory management that shares SPM resources of many-core platforms based on the memory requirement of applications. Onchip SPM resources, which form a pool of SPMs, are mapped to executing applications by a runtime manager.
- Auction-based memory management: A distributed scheme to improve scalability of SPMPool approach. Many-core platform is divided into different regions. The SPM pool of each region can expand or shrink using a distributed auction mechanism.
- Memory phases: A distinction from program phases which can capture temporal change in memory requirement of a single application. We proposed different offline and online mechanisms to detect and use memory phases in many-cores.

5.2 Future Research

The runtime management schemes proposed in this dissertation can be expanded in different directions:

- To exploit the benefits of SPMs and caches, their co-existence is very likely in future architectures. It may prove helpful to study the interplay of cache and SPM to find mechanisms that maximize the power and performance benefits of the system when both resources are available on the platform.

- In the auction-based distributed memory management, we assumed that task mapping is fixed and the main focus was on memory management. There is potential to enhance the overall performance of the system by incorporating task mapping, task migration, and memory management.

- We simulated the proposed schemes in this thesis at the memory trace level. A full system implementation of these schemes will provide more details understanding of the actual overhead of these methods on the system. A detailed implementation of SPM and operating system support for SPM is an ongoing subject of research.

- Although memory phases are used to manage SPM mapping in this thesis, the concept of memory phases can be applied to other domains of memory management such as data prefetching or cache management. A detailed study of memory phases in these domains would be an interesting research topic.

Bibliography

[1] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: verifying memory coherence in the cray x1. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003.

[2] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel. mrts: Run-time system for reconfigurable processors with multi-grained instruction-set extensions. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, 2011.

[3] W. Ahmed, M. Shafique, L. Bauer, and J. Karlsruhe. Adaptive resource management for simultaneous multitasking in mixed-grained reconfigurable multi-core processors. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on*, 2011.

[4] I. Anagnostopoulos, A. Bartzas, G. Kathareios, and D. Soudris. A divide and conquer based distributed run-time mapping methodology for many-core platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, 2012.

[5] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris. Distributed run-time resource management for malleable applications on many-core platforms. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, 2013.

[6] F. Bachmann and L. Bass. Managing variability in software architectures. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, 2001.

[7] K. Bai and A. Shrivastava. Automatic and efficient heap data management for limited local memory multicore architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013.

[8] K. Bai, A. Shrivastava, and S. Kudchadker. Stack data management for limited local memory (llm) multi-core processors. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, 2011.

[9] R. Balasubramonian et al. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.

[10] M. Banikazemi, D. Poff, and B. Abali. Pam: a novel performance/power aware meta-scheduler for multi-core systems. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, 2008.

[11] L. A. D. Bathen, N. D. Dutt, D. Shin, and S.-S. Lim. Spmvisor: Dynamic scratchpad memory virtualization for secure, low power, and high performance distributed on-chip memories. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2011.

[12] N. Beckmann and D. Sanchez. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[13] P. Bellasi, G. Massari, and W. Fornaciari. A rtrm proposal for multi/many-core platforms and reconfigurable applications. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, 2012.

[14] D. P. Bertsekas. A distributed algorithm for the assignment problem". In *Lab. for Information and Decision Systems Working Paper, M.I.T.,Cambridge, MA.*, 1979.

[15] D. P. Bertsekas. Auction algorithms. In *Encyclopedia of Optimization*, 2002.

[16] D. P. Bertsekas and D. A. Castanon. Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Computing*, 1991.

[17] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[18] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. Romero, and C. Scordino. Resource management on multicore systems: The actors approach. *IEEE Micro*, 2011.

[19] N. Binkert, B. Beckmann, and Black. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[20] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. *SIGOPS Oper. Syst. Rev.*, 1989.

[21] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[22] R. Braithwaite, P. McCormick, and W.-c. Feng. Empirical memory-access cost models in multicore numa architectures. *Virginia Tech Department of Computer Science*, 2011.

[23] J. A. Carballo, W. T. J. Chan, P. A. Gargini, A. B. Kahng, and S. Nath. Itrs 2.0: Toward a re-framing of the semiconductor technology roadmap. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014.

[24] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing*, 2007.

[25] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, 2002.

[26] W. Che, A. Panda, and K. Chatha. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010.

[27] C.-B. Cho and T. Li. Complexity-based program phase analysis and classification. In *PACT*, 2006.

[28] D. Cho, S. Pasricha, I. Issenin, N. Dutt, M. Ahn, and Y. Paek. Adaptive scratch pad memory management for dynamic behavior of multimedia applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2009.

[29] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[30] J. Choi, H. Oh, S. Kim, and S. Ha. Executing synchronous dataflow graphs on a spm-based multicore architecture. In *Proceedings of the 49th Annual Design Automation Conference*, 2012.

[31] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[32] G. Cormode and M. Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 2010.

[33] H. Corporaal. Ttas: Missing the ilp complexity wall. *J. Syst. Archit.*, 1999.

[34] Y. Cui, W. Zhang, and H. Yu. Decentralized agent based re-clustering for task mapping of tera-scale network-on-chip system. In *2012 IEEE International Symposium on Circuits and Systems*, 2012.

[35] A. Das and D. Grosu. Combinatorial auction-based protocols for resource allocation in grids. In *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[36] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Lger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, 2013.

[37] N. Deng, W. Ji, J. Li, and Q. Zuo. A semi-automatic scratchpad memory management framework for cmp. In *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, 2011.

[38] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. *SIGARCH CAN*, 2002.

[39] S. Dighe, S. R. Vangal, P. A. Aseron, S. Kumar, T. Jacob, K. A. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V. K. De, and S. Borkar. Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor. *J. Solid-State Circuits*, 2011.

[40] Dropsho et al. Integrating adaptive on-chip storage structures for reduced dynamic power. In *PACT*, 2002.

[41] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 2009.

[42] N. Edalat, C.-K. Tham, and W. Xiao. An auction-based strategy for distributed task allocation in wireless sensor networks. *Computer Communications*, 2012.

[43] B. Egger, J. Lee, and H. Shin. Scratchpad memory management in a multitasking environment. In *Proceedings of the 8th ACM International Conference on Embedded Software*, 2008.

[44] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 2011.

[45] L. Fan, P. Trinder, and H. Taylor. Deadline-driven auctions for npc host allocation in p2p mmogs. *Int. J. Adv. Media Commun.*, 2010.

[46] M. A. A. Faruque, R. Krist, and J. Henkel. Adam: Run-time agent-based distributed application mapping for on-chip communication. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, 2008.

[47] F. Fazzino, M. Palesi, and D. Patti. Noxim: Network-on-chip simulator. *URL: http://sourceforge. net/projects/noxim*, 2008.

[48] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Annual Design Automation Conference*, 2004.

[49] X. Fu and X. Wang. Utilization-controlled task consolidation for power optimization in multi-core real-time systems. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2011.

[50] L. Gauthier, T. Ishihara, H. Takase, H. Tomiyama, and H. Takada. Minimizing inter-task interferences in scratch-pad memory usage for reducing the energy consumption of multi-task systems. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2010.

[51] Y. Ge, P. Malani, and Q. Qiu. Distributed task migration for thermal management in many-core systems. In *Proceedings of the 47th Design Automation Conference*, DAC '10, 2010.

[52] J. Gomoluch and M. Schroeder. Market-based resource allocation for grid computing: A model and simulation. In *Middleware Workshops*, 2003.

[53] G. Guan, C. Ma, and J. Wu. *Hierarchical Clustering Techniques*, chapter 7, pages 109–149. SIAM, 2007.

[54] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, S. Swanson, and D. Sylvester. Underdesigned and opportunistic computing in presence of hardware variability. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 2013.

[55] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, 2001.

[56] G. Hamerly et al. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Journal of Instruction Level Parallelism*, 2005.

[57] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[58] M. A. Heinrich. *The performance and scalability of distributed shared-memory cache coherence protocols*. dissertation, Stanford University, 2002.

[59] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hbner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. Invasive manycore architectures. In *17th Asia and South Pacific Design Automation Conference*, 2012.

[60] J. Henkel, V. Narayanan, S. Parameswaran, and J. Teich. Run-time adaption for highly-complex multi-core systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, 2013.

[61] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[62] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[63] H. Hoffman. *Seec: A Framework for Self-aware Management of Goals and Constraints in Computing Systems (Power-aware Computing, Accuracy-aware Computing, Adaptive Computing, Autonomic Computing)*. PhD thesis, Massachusett Institute of Technology, 2013.

[64] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and

T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 2010.

[65] T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *PACT*, 2006.

[66] IBM. The cell project. `http://researcher.watson.ibm.com/`, 2005.

[67] IBM. IBM ILOG AMPL. `http://ampl.com/`, 2010.

[68] Intel. Single-chip cloud computer. `http://intel.com/`, 2009.

[69] Intel Lab. *The SCC programmers guide*, March 2014.

[70] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO*, 2006.

[71] ITRS. Executive report. `http://www.itrs2.net/`, 2015.

[72] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2007.

[73] A. Jaleel. Memory Characterization of Workloads Using Instrumentation-Driven Simulation A Pin-based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites. Technical report, Intel, 2007.

[74] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[75] W. Ji, N. Deng, F. Shi, Q. Zuo, and J. Li. Dynamic and adaptive spm management for a multi-task environment. *J. Syst. Archit.*, 2011.

[76] D. Kaseridis, J. Stuecheli, and L. John. Bank-aware dynamic cache partitioning for multicore architectures. In *Parallel Processing, 2009. ICPP '09. International Conference on*, 2009.

[77] P. Klemperer. Auction theory: A guide to the literature. *Journal of Economic Surveys*, 1999.

[78] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE micro*, 2008.

[79] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel. Distrm: Distributed resource management for on-chip many-core systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, 2011.

[80] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, 2005.

[81] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *HPCA*, 2005.

[82] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[83] H. Lee, S. Cho, and B. Childers. Stimuluscache: Boosting performance of chip multiprocessors with excess cache. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010.

[84] H. Lee, S. Cho, and B. Childers. Cloudcache: Expanding and shrinking private caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011.

[85] T. Li and L. K. John. Adirpnb: A cost-effective way to implement full map directory-based cache coherence protocols. *IEEE Trans. Comput.*, 2001.

[86] Y. Li, Z. Gao, Y. Yang, Z. Guan, X. Chen, and X. Qiu. A cluster-based negotiation model for task allocation in wireless sensor network. In *2010 International Conference on Network and Service Management*, 2010.

[87] W. Y. Lin, G. Y. Lin, and H. Y. Wei. Dynamic auction mechanism for cloud resource allocation. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010.

[88] L. Liu and D. A. Shell. Optimal market-based multi-robot task allocation via strategic pricing. In *Robotics: Science and Systems*, 2013.

[89] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. University of California Press, 1967.

[90] A. Marongiu and L. Benini. An openmp compiler for efficient use of distributed scratch-pad memory in mpsocs. *Computers, IEEE Transactions on*, 2012.

[91] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, 2010.

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

[93] O. Mutlu. Memory systems in the many-core era: Challenges, opportunities, and solution directions. In *Proceedings of the International Symposium on Memory Management*, 2011.

[94] O. Mutlu. Memory scaling: A systems architecture perspective. In *2013 5th IEEE International Memory Workshop*, pages 21–25, 2013.

[95] S. R. Nassif. Modeling and analysis of manufacturing variations. In *Custom Integrated Circuits, 2001, IEEE Conference on.*, 2001.

[96] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Design, Automation and Test in Europe*, 2005.

[97] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. Sdrm: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Proceedings of the 15th International Conference on High Performance Computing*. Springer Berlin Heidelberg, 2008.

[98] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European Conference on Design and Test*, 1997.

[99] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[100] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[101] P. Ratanaworabhan and M. Burtscher. Program phase detection based on critical basic block transitions. In *ISPASS*, 2008.

[102] K. Rupp. 40 years of microprocessor trend data. `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data`. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten. New plot and data collected for 2010-2015 by k. Rupp.

[103] G. Sabin, M. Lang, and P. Sadayappan. Moldable parallel job scheduling using job efficiency: An iterative approach. In *Proceedings of the 12th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'06, 2007.

[104] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing*, 2008.

[105] J. Sartori and R. Kumar. Distributed peak power management for many-core architectures. In *2009 Design, Automation Test in Europe Conference Exhibition*, 2009.

[106] A. Sembrant et al. Phase guided profiling for fast cache modeling. In *CGO*, 2012.

[107] M. Shafique, L. Bauer, W. Ahmed, and J. Henkel. Minority-game-based resource allocation for run-time reconfigurable multi-core processors. In *Design, Automation and Test in Europe Conference and Exhibition, 2011. Proceedings*, 2011.

[108] A. Sharifi, S. Srikantaiah, M. Kandemir, and M. Irwin. Courteous cache sharing: Being nice to others in capacity management. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012.

[109] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. *SIGPLAN Not.*, 2004.

[110] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.

[111] A. Shrivastava, N. Dutt, J. Cai, M. Shoushtari, B. Donyanavard, and H. Tajik. Automatic management of software programmable memories in many-core architectures. *IET Computers Digital Techniques*, 10(6), 2016.

[112] T. Sondag and H. Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *CGO*, 2011.

[113] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: Managing shared caches in chip multiprocessors. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[114] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002.

[115] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[116] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad allocation for concurrent embedded software. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2008.

[117] A. Sulistio and R. Buyya. A time optimization algorithm for scheduling bag-of-task applications in auction-based proportional share systems. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, 2005.

[118] J. Sun, E. Modiano, and L. Zheng. Wireless channel allocation using an auction algorithm. *IEEE Journal on Selected Areas in Communications*, 2006.

[119] H. Tajik, B. Donyanavard, and N. Dutt. Detecting and Using Memory Phases. Technical report, CECS TR 16-05, University of California, Irvine, 2016.

[120] H. Tajik, B. Donyanavard, and N. Dutt. On detecting and using memory phases in multimedia systems. In *Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia*, ESTIMedia'16, 2016.

[121] H. Tajik, B. Donyanavard, J. Jahn, J. Henkel, and N. Dutt. Spmpool: Runtime spm management for memory-intensive applications in embedded many-cores. *ACM Trans. Embed. Comput. Syst.*, 2016.

[122] H. Takase, H. Tomiyama, and H. Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010.

[123] Techcrunch. Digital media consumption report. `https://techcrunch.com/`, 2015.

[124] Tilera. gx family. `http://www.tilera.com/`, 2010.

[125] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 2006.

[126] R. F. van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 2011.

[127] M. Verma, S. Steinke, and P. Marwedel. Data partitioning for maximal scratchpad usage. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, 2003.

[128] J. D. Warnock, J. M. Keaty, J. Petrovick, J. G. Clabes, C. J. Kircher, B. L. Krauter, P. J. Restle, B. A. Zoric, and C. J. Anderson. The circuit and physical design of the power4 microprocessor. *IBM Journal of Research and Development*, 2002.

[129] A. Weichslgartner, S. Wildermann, and J. Teich. Dynamic decentralized mapping of tree-structured applications on noc architectures. In *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, 2011.

[130] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 1995.

[131] Y. Xie and G. H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[132] M. M. Zavlanos, L. Spesivtsev, and G. J. Pappas. A distributed auction algorithm for the assignment problem. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, 2008.

[133] L. Zhang, M. Qiu, W.-C. Tseng, and E. H.-M. Sha. Variable partitioning and scheduling for mpsoc with virtually shared scratch pad memory. *Journal of Signal Processing Systems*, 2010.

[134] Y. Zhang, D. Niyato, and P. Wang. An auction mechanism for resource allocation in mobile cloud computing systems. In *Proceedings of the 8th International Conference on Wireless Algorithms, Systems, and Applications*, 2013.

[135] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2011.

[136] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[137] R. M. Zlot. *An auction-based approach to complex task allocation for multirobot teams.* PhD thesis, Georgia Institute of Technology, 2006.