An Adaptive Framework for Managing Heterogeneous Many-Core Clusters

Muhammad Mustafa Rafique

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Ali R. Butt, Chair
Kirk W. Cameron
Wu Feng
Leyla Nazhandali
Dimitrios Nikolopoulos
Eli Tilevich

September 22, 2011
Blacksburg, Virginia, USA

Keywords: Heterogeneous Computing, Programming Models, Resource Management and Scheduling, Resource Sharing, High-Performance Computing

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

An Adaptive Framework for Managing Heterogeneous Many-Core Clusters

Muhammad Mustafa Rafique

ABSTRACT

The computing needs and the input and result datasets of modern scientific and enterprise applications are growing exponentially. To support such applications, High-Performance Computing (HPC) systems need to employ thousands of cores and innovative data management. At the same time, an emerging trend in designing HPC systems is to leverage specialized asymmetric multicores, such as IBM Cell and AMD Fusion APUs, and commodity computational accelerators, such as programmable GPUs, which exhibit excellent price to performance ratio as well as the much needed high energy efficiency. While such accelerators have been studied in detail as stand-alone computational engines, integrating the accelerators into large-scale distributed systems with heterogeneous computing resources for data-intensive computing presents unique challenges and trade-offs. Traditional programming and resource management techniques cannot be directly applied to many-core accelerators in heterogeneous distributed settings, given the complex and custom instruction sets architectures, memory hierarchies and I/O characteristics of different accelerators. In this dissertation, we explore the design space of using commodity accelerators, specifically IBM Cell and programmable GPUs, in distributed settings for data-intensive computing and propose an adaptive framework for programming and managing heterogeneous clusters.

The proposed framework provides a MapReduce-based extended programming model for heterogeneous clusters, which distributes tasks between asymmetric compute nodes by considering workload characteristics and capabilities of individual compute nodes. The framework provides efficient data prefetching techniques that leverage general-purpose cores to stage the input data in the private memories of the specialized cores. We also explore the use of an advanced layered-architecture based software engineering approach and provide mixin-layers based reusable software components to enable easy and quick deployment of heterogeneous clusters. The framework also provides multiple resource management and scheduling policies under different constraints, e.g., energy-aware and QoS-aware, to support executing concurrent applications on multi-tenant heterogeneous clusters. When applied

to representative applications and benchmarks, our framework yields significantly improved performance in terms of programming efficiency and optimal resource management as compared to conventional, hand-tuned, approaches to program and manage accelerator-based heterogeneous clusters.

Dedicated to my parents, and wife,
for their endless love, encouragement, and support ...

Acknowledgments

I would like to thank my adviser, Ali R. Butt, for his continuous guidance, valuable advice, and endless encouragement during my graduate studies at Virginia Tech. When I look back over the past four years, I find myself very fortunate to have him as my adviser. He has always been a continuous source of help and motivation to me, both inside and outside of Virginia Tech. Without his support, the successful completion of this work could have been an uphill task. I would also like to thank the members of my advisory committee; Kirk W. Cameron, Wu Feng, Leyla Nazhandali, Dimitrios Nikolopoulos, and Eli Tilevich, for providing me with their valuable feedback. I would like to express special thanks to Dr. Nikolopoulos for posing challenging research problems and helping me in setting up the initial directions for my graduate research. He is indeed a remarkable researcher, meticulous academician, and above all a very pleasant personality to work with. I would also like to express my deepest gratitude to Dr. Tilevich for all the pair writing sessions, for teaching me the art of selling the research achievements, and for always being there for valuable discussions.

I was supported by a Fulbright Scholarship during my Ph.D. studies. I would like to thank the Institute of International Education, and the Fulbright commission for providing me with all the financial assistance during these years. A significant part of this work has been done at NEC Labs, where I was exposed to the real-world research challenges that are faced in the industry. I would like to thank Srihari Cadambi, Niskham Ravi, and Srimat Chakradhar at NEC Labs for overseeing my research, and for providing me with all the required equipment and support.

I would like to thank my colleagues at Distributed Systems and Storage Laboratory (DSSL), especially Henry Monti, Guanying Wang, Min Li, Puranjoy Bhattacharjee, and Aleksandr Khasymski, for their valuable discussions and feedbacks, and for all the fun that we have had together while attending different conferences throughout these years. I would also like to

thank my friends at Virginia Tech, especially Zaki Malik, Imran Akhtar, Shakeel Nasir, Zeshan Hyder, Syed Amaar Ahmad, Syed Makhmoor Mazahir, and Mohammed Rabius Sunny for making my stay at Virginia Tech a memorable one.

A very special thanks goes to my sisters; Aisha Rashid, Aqsa Rehan, and Sana Rafique, for their passionate love and endless support towards me. They have always made me feel like a special one, have always saved the best cut of meat for me, and have always helped me in any way possible.

Finally, I would like to thank the special ones; my mother (Firdous Rafique), my father (Muhammad Rafique), and my wife (Abeer Mustafa). Although no words can express my profound gratitude towards them, I still have to give it a shot. I would like to thank my parents for providing me with all the support, love and encouragements, since my childhood. Their endless determination and unshakable faith have always been a driving force for me in my pursuit of graduate studies. I would like to thank Abeer for staying up many nights with me while I worked on this dissertation, taking over the family responsibilities and managing the household, and for everything that she has done for me during all these years.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AMP	Asymmetric Multicore Processors
API	Application Programming Interface
APU	Accelerated Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
EIB	Element Interconnect Bus
FCFS	First Come First Served
FOP	Feature-Oriented Programming
FPGA	Field-Programmable Gate Array
GPGPU	General-Purpose computing on Graphics Processing Units
GPP	General-Purpose Processors
GPU	Graphics Processing Unit
HPC	High Performance Computing

I/O	Input/Output
ISA	Instruction Set Architecture
LOC	Lines of Code
MPI	Message Passing Interface
NFS	Network File System
PCI	Peripheral Component Interconnect
PPE	Power Processor Element
PS3	PlayStation 3
QoS	Quality of Service
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
SDK	Software Development Kit
SIMD	Single-Instruction Multiple-Data
SIMT	Single-Instruction Multiple-Threads
SMT	Simultaneous Multi-Threading
SPE	Synergistic Processing Elements
SPMD	Single-Program Multiple Data
TBB	Threading Building Blocks
TCO	Total Cost of Ownership

Chapter 1

Introduction

Large-scale computing systems comprising thousands of cores routinely serve as workhorses for enabling scientific discoveries and enterprise activities. With the increase in the size of an average computing cluster and the requirements to orchestrate and manage large input and output datasets, the use of computational accelerators, coprocessors, accelerated processing units (APUs) and asymmetric multicore processors (AMPs) with a mix of general-purpose processors (GPP) and specialized cores have become a leading paradigm for cost-effective high-end computing setups. Processor vendors will soon deliver Teraflop-capable, single-chip multi-processors, by integrating many simple cores with SIMD (Single-Instruction Multiple-Data) or SIMT (Single-Instruction Multiple-Threads) datapaths. Cell processors [1,2], AMD Fusion APUs [3], programmable graphics processing units (GPUs) from NVIDIA [4, 5], Larrabee [6] and the Single-chip Cloud Computer [7] are representatives of this class of processors with significant market interest and demonstrated potential [8–14]. Teraflop-capable processors accelerate computation by design, while staying within a reasonable cost budget. Therefore, they are considered as one of the most promising computational workhorses [5,15–22] for High Performance Computing (HPC). Using heterogeneous systems that combine fine-grain parallelism with coarse-grain parallelism using ten or thousands of processors has helped improved the performance of HPC applications [23–25]. As a result, these heterogeneous architectures are becoming more popular for consumer applications including computer gaming, multimedia applications, desktop computing, HPC, robotics and embedded applications.

Using heterogeneous many-core accelerators and coprocessors in a large-scale distributed setting is different from using traditional multicore processors that are composed of homogeneous cores having same instruction set architecture (ISA). Although these heterogeneous many-core processors offer better computational performance and power efficiency as com-

pared to the traditional multicores, their use in a large-scale cluster introduces additional design and run-time challenges. This dissertation proposes an adaptive framework for managing heterogeneous many-core clusters, and presents novel solutions to the programming and resource-management challenges that are associated with the use of heterogeneous many-core processors in large-scale computing clusters.

In this chapter, we provide the necessary background for understanding the research done in this dissertation. In particular, Section 1.1 provides an overview of the challenges and issues related to the use of heterogeneous many-core systems that we attempt to address on multicores. Section 1.2 highlights the objective of this research. Section 1.3 summarizes the research contributions that we make in this dissertation. Section 1.4 provides an outline of the remainder of this dissertation.

1.1 Challenges in Using Heterogeneous Many-Core

The new paradigm shift from homogeneous multicores to heterogeneous many-core processors in a node as well as in a large-scale cluster introduces many challenges, with respect to programmability, I/O orchestration, and resource management. Many-core architectures provide concurrency, however, in order to take advantage of the available hardware resources, new programming paradigms, intelligent data-distribution techniques, and efficient resource management schemes need to be developed to exploit the capabilities of heterogeneous cores to their full extent [26,27]. In the following we provide an overview of the challenges involved in using heterogeneous systems that we attempt to address in this dissertation.

1.1.1 Programming Heterogenous Many-Core Systems

Programming asymmetric systems, even at a single-chip level, is not very well understood, and is the topic of much research [28–32]. In contrast, programming models for symmetric clusters, e.g., MPI [33] and SHMEM [34], are mature, user-friendly, and free the application programmers from low-level system management details. Programming accelerators-based heterogeneous systems requires working with multiple ISA and multiple compilation targets. Lack of easy-to-use and efficient programming models and tools to model the system heterogeneity is one of the key challenges in programming heterogeneous clusters. Furthermore, absence of sophisticated debuggers and profiling support to better understand the application behavior also imposes a barrier for using heterogeneous resources in mainstream

application development.

While hiding architectural asymmetry and system scale from parallel programming model are desirable properties [35], they are challenging to implement in a heterogeneous system, where exploiting the customization and computational density of each computational unit is a first-order consideration. Mapping a high-level parallel programming model to accelerators while hiding the details of accelerator hardware is extremely challenging and even undesirable if raising the level of abstraction comes at a performance cost. Current approaches for programming accelerator-based heterogeneous clusters are either ad hoc or specific to an installation [21, 36], thus posing several challenges when applying to general setups. Further challenges arise because of heterogeneity, which manifest both in the programming model and in resource management. Implementing an accessible programming model on systems with many cores, many processors, multiple ISAs and multiple compilation targets requires drastic modifications of the entire software stack. Suitable programming models that adapt to the varying capabilities of the accelerator-type components have not been developed yet. This deficit forces application writers who want to use accelerators on clusters to micro-manage installation-specific resources. Using architecture-specific solutions is highly undesirable, as it compromises productivity, portability, and sustainability of the involved systems and applications.

The effects of alternative workload distributions between general-purpose processors and accelerators are also not well-understood. Accelerators typically have much higher compute density and raw performance than conventional processors, therefore coupling accelerators with conventional processors may introduce imbalance between the two. Accelerators also typically have limited capabilities for managing external system resources, such as communication and I/O devices, thus requiring support from general-purpose processors and special consideration while designing the I/O management software. To ensure overall high efficiency, I/O management on accelerator-based systems needs to carefully orchestrate data transfers and work distribution between heterogeneous components.

1.1.2 Data Prefetching in Heterogeneous Many-Core Systems

Modern many-core accelerators follow a design philosophy that favors many simple cores with wide datapaths. This leads to more power-efficient designs while sustaining exponential improvement of performance. Unfortunately, this approach strips processors from capabilities of executing control-intensive code efficiently, such as branch predictors, multiple-issue,

dynamic instruction scheduling, and speculation. Therefore, accelerators can not support efficiently control-intensive code, such as most operating system services. Furthermore, although accelerators have high compute density, they also have limited storage space directly accessible to them on-chip or via a dedicated link. Therefore, any programming model that integrates accelerators needs to implement efficient data transfers to and from the on-chip memory of the accelerators. Scaling the resource heterogeneity beyond a single node requires synthesis of parallel execution and data communication techniques. These techniques should balance the computational speed of accelerators, with the limited off-chip and off-node data transfer bandwidth. Achieving this goal while retaining properties that help boost programmer productivity, such as being agnostic of the number and architecture of processors, is a major challenge.

Asymmetric multicores and coprocessors have been primarily used for compute-intensive workloads. However, modern applications often comprise of I/O-intensive phases that process large data in streams, e.g., compression/decompression, encryption/decryption, video processing, and workflow-based scientific computing. The compute kernels of such applications can benefit from asymmetric clusters, but the inherent mismatch between the heterogeneous components poses a major obstacle to sustaining the high I/O rates necessary for reaping these benefits. This poses critical research challenges for symmetric as well as asymmetric clusters that should be addressed to fully realize the capabilities of available resources [37–39]. The limited attached storage of an accelerator-based resource may prevent it from hosting the suite of essential I/O optimizations, e.g., caching, prefetching, etc., which are standard in general-purpose nodes, and force them to rely on GPPs to prefetch data and perform I/O on their behalf. Furthermore, unnecessary I/O serialization may occur at the cluster manager as it tries to feed multiple accelerators. Given that any decent-sized asymmetric cluster will employ multiple nodes, performance-degrading contention may occur for storage and network resources. Previously developed data distribution and task management libraries for asymmetric accelerator-based architectures [40], delegate parameterization of data transfers and workload scheduling to the programmer. While this is a workable approach, it poses additional design challenges to the programmer and increases the application development time. Effective use of heterogeneous accelerators and coprocessors in a distributed setting requires creating efficient and transparent I/O management, prefetching, and data staging techniques that hide the underlying resource asymmetry and heterogeneity from the programmer while providing the desired performance.

1.1.3 Managing Heterogeneous Many-Core Systems

While the potential of many-core systems to catalyze HPC systems and data centers is clear, attempting to integrate heterogeneous many-core processors seamlessly in large-scale computing installations raises several challenges in managing these resources. There is an inherent imbalance between general-purpose cores, accelerators and coprocessors in asymmetric settings. The trend towards integrating relatively simple cores with extremely efficient vector and stream processing units leads to designs that are inherently compute-efficient but control-inefficient. General-purpose cores are efficient in executing control-intensive code, therefore they tend to be employed primarily as controllers of parallel execution and communication with accelerators. Accelerators on the other hand, are efficient in executing data-parallel computational tasks. To address this problem, large-scale system installations use ad hoc approaches to pair accelerators with more control-efficient processors, such as x86 multicore CPUs [21], whereas processor architecture moves in the direction of integrating control-efficient and compute-efficient cores on the same chip [6].

Another fundamental resource management challenge arise in a multi-tenant heterogenous setup, such as cloud computing environment, where heterogeneous resources are shared between multiple concurrently executing applications. Coprocessors and accelerators generally do not support executing multiple applications concurrently, thus require especial scheduling considerations while supporting multi-tenancy. In an effort to increase concurrency between multiple kernels, the latest NVIDIA GPUs now support executing multiple kernels concurrently, however, these concurrent kernels must be launched from the same process. This support is insufficient to support executing multiple applications concurrently. Furthermore, each computing resource offers a unique performance advantages for different applications, and using heterogeneous coprocessors and accelerators in a single cluster supporting multi-tenancy introduces additional challenges of optimal application to coprocessor mapping to improve the overall system performance.

One of the key benefits that attract the use of heterogeneous many-core processors in data centers and enterprise setups is the power-efficiency offered by these resources. Using high performance energy-efficient coprocessors coupled with low performance energy-efficient general-purpose processors is a viable approach that can be adapted to minimize the energy consumption of a computing cluster. However, lack of effective power-aware resource management and scheduling techniques that can be applied to heterogeneous multi-tenant

clusters in general reduces the power-efficiency benefits of these resources. Developing efficient energy-aware resource management and scheduling techniques is a challenging task since it requires information about the energy profile of all the heterogeneous computing resources and the characteristics of the concurrent applications that are executed on the cluster. Inefficient scheduling and management of these power-efficient resources results in more energy consumption and void the purpose of their use.

1.2 Research Objectives

This dissertation focuses on developing productive programming models and adaptive resource management framework for heterogeneous many-core clusters for compute and data intensive computing. While designing the framework, we propose innovative solutions and adapt node and cluster level middlewares to emerging heterogeneous systems. The main objectives of this research are to:

1. design and develop efficient programming models and abstractions that can hide the underlying architectural asymmetry of the heterogeneous clusters from the application programmer and improve programming efficiency;

2. design and develop efficient data distribution, I/O management, and prefetching techniques that can be used to eliminate the memory limitations of accelerator-based systems;

3. design and develop novel scheduling mechanisms that can be used to improve the energy-efficiency of heterogeneous clusters, and enable them to support concurrent applications in a multi-tenant environment.

The scope of the research presented in this dissertation extends across system software and hardware architectures in three specific areas: programming models, memory subsystems, and resource scheduling. We use commodity off-the-shelf accelerators, specifically IBM Cell Broadband Engine [1] and NVIDIA GPUs [4] as computational accelerators and coprocessors in various heterogeneous configurations in this dissertation. From application front, we target various scientific and enterprise applications with the specific focus on application efficiency and high performance computing.

1.3 Research Contributions

This dissertation proposes an adaptive framework for programming and managing heterogeneous many-core clusters. While designing the proposed framework, we create innovative solutions to address the challenges of programmability and resource management in heterogeneous clusters. In the following, we highlight specific research contributions that we make in this dissertation.

1. We propose an extended MapReduce-based programming model [41–43] to program Cell and GPU-based heterogeneous clusters. Our approach utilizes all available resources irrespective of the resource heterogeneity, and hide the inherent resource asymmetry from the application programmer. We evaluate our approach using Cell and GPU-based accelerators and show that the proposed programming model can be used efficiently for a variety of conventional and hierarchical cluster configurations and scales well with the number of compute nodes.

2. We investigate the use of advanced component-based software engineering approach [44] to program accelerator-based heterogeneous systems. In particular, we develop mixin-layers based reusable software components for accelerator-based heterogeneous clusters, and effectively encapsulate both reusable and custom feature implementations within separate components. In our design, custom implementations–both heterogeneous and platform specific–can be introduced as new components, which can then be plugged-into our component architecture of a given accelerator-based cluster configuration. We evaluate our approach using Cell and GPU-based heterogeneous cluster and show that the layered architecture helped significantly reduce the amount of code that had to be changed to support heterogeneous nodes, and the majority of components could actually be reused as is.

3. We investigate several I/O prefetching techniques [45] for Cell processor, and propose an asynchronous prefetching approach that prefetches the required data using the general-purpose core and then offloads it to the specialized cores using decentralized DMA operations, rather than letting the specialized cores to do the I/O directly. We evaluate the proposed scheme and show that it significantly breaks up the I/O bottleneck and results in better performance for data-intensive workloads compared to the case where all I/O is handled at the specialized core. This work enables the use of

asymmetric multicores, such as Cell/BE, for executing I/O intensive workloads in a heterogeneous cluster with minimal execution stalls and improve system performance.

4. We propose various I/O management techniques [41, 46] for heterogeneous clusters and integrate them in our extended MapReduce programming model. Specifically, to close the computation-I/O gap between heterogeneous resources, we explore a dynamic streaming approach to transfer large data between cluster resources based on the memory capacities of communicating components. We also investigate a capability-aware workload distribution technique that maps the concurrent workloads on the compute resources based on the workload requirements and the computation and I/O capabilities of compute resources. We evaluate the proposed approaches and show that they significantly improve system performance.

5. We propose a novel power-aware resource management and scheduling scheme [47] for heterogeneous clusters that can be incorporated into the proposed framework to maximize the work done per watt and reduce the total the cost of ownership for computing clusters. The proposed scheduling scheme identifies the optimal cluster configuration, i.e., homogeneous or heterogeneous, based on the power profiles of the deployed servers and the characteristics of the workloads, and maximizes work done per watt by dynamically assigning low power states to servers based on the current request rate. We evaluate the proposed scheduling scheme and show significant improvements in the energy efficiency of a heterogeneous cluster.

6. We propose a QoS-aware scheduling scheme [48] that can be incorporated into the proposed framework to enable efficient dynamic sharing of a heterogeneous clusters across multiple concurrently-executing applications, each with arbitrary load spikes under strict performance requirements. The proposed scheme monitors the load on each application, collects past performance data, and dynamically builds simple performance models using available processing resources. Based on the performance model, it computes priority for pending user requests and reorders them across different applications to achieve the desired performance requirements. We evaluate the proposed approach in a multi-tenant environment and show that it significantly improves the QoS requirements of concurrent applications.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2, we provide an overview of the related work and the background technologies that lay the foundation of the research conducted in this dissertation. In Chapter 3, we describe how MapReduce can be extended to program heterogeneous clusters comprising accelerator-based compute nodes, and present an extended MapReduce-based programming model to program Cell and GPU-based heterogeneous clusters. Furthermore, we also explore layered-architecture based advanced software engineering approach to program heterogeneous clusters, and describe how mixin-layers based reusable software components can be used to enable the quick deployment of heterogeneous clusters. In Chapter 4, we present how prefetching techniques can be used to enable the use of asymmetric multicores for data-intensive computing, and develop prefetching techniques to eliminate the I/O bottlenecks from the Cell-based asymmetric multicores. In Chapter 5, we present the design of a capability-aware workload distribution technique that enables executing concurrent applications on the heterogeneous resources by considering the I/O characteristics of individual applications and the I/O capabilities of the available compute nodes. In Chapter 6, we explore how energy-aware workload scheduling can be used to maximize the energy benefits of heterogeneous resources. In Chapter 7, we explore how QoS-aware resource scheduling can be used to meet the performance requirements of concurrent applications, and present a QoS-aware resource scheduling mechanism for multi-tenant heterogeneous clusters. Finally, Chapter 8 concludes this dissertation.

Chapter 2

Background and Related Work

In this chapter, we present an overview of the technologies that enable the research conducted in this dissertation. We also discuss the closely related work on programming models, data staging and prefetching techniques, and resource management schemes.

2.1 Enabling Technologies

In the following subsections, we provide an introduction of the heterogeneous accelerators, specifically IBM Cell processor and CUDA-enabled GPUs, that we have used in this dissertation. We also provide an overview of the MapReduce programming model that we have adapted to program heterogeneous clusters.

2.1.1 Many-Core Computational Accelerators

Multicore processors with tightly coupled accelerators are becoming common, with the potential to sustain supercomputer-class node performance for dense computations, under a reasonable power budget. Large-scale data centers typically employ commodity off-the-shelf components to yield a cost-efficient setup. While commodity hardware has been common place in HPC setups for almost two decades, large-scale data centers of commercial interest, e.g., Google [49], Amazon's EC2 [36], etc., follow the same approach to building efficient systems at scale.

Availability of commodity accelerators such as the Cell [1, 2, 50] in the Cell-based Sony PlayStation 3 (PS3) [51, 52], and NVIDIA GPU-based graphics engines [53, 54] renders these computational engines prime candidates for deployment in large-scale HPC systems. The rapid growth in high-speed networks and the use of low cost commodity off-the-shelf hardware, makes such distributed asymmetric clusters natural substitutes for expensive high-end

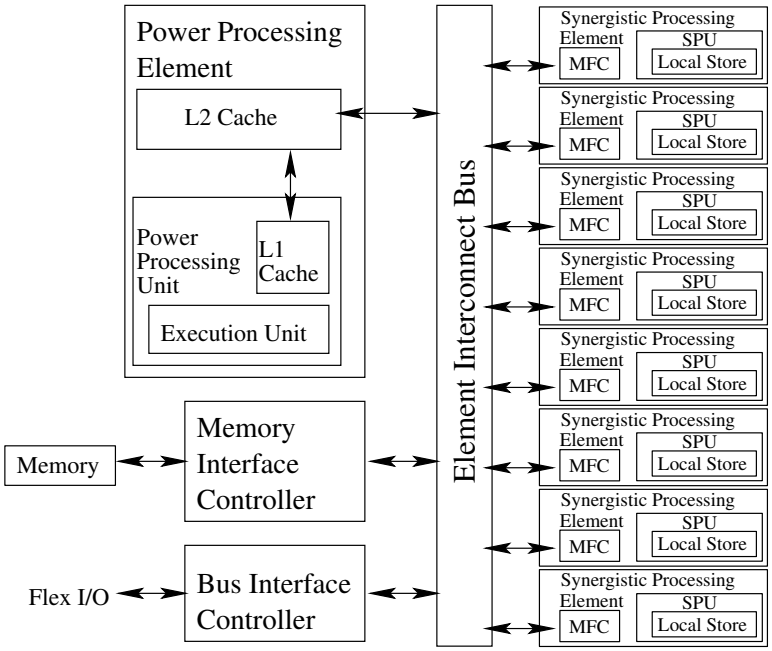


Figure 2.1 Cell Broadband Engine system architecture.

supercomputers. The use of off-the-shelf components in large-scale clusters has been demonstrated both in academia, e.g., Condor [55], and industry, e.g., Tianhe [56], Google [49], Roadrunner [21] and Amazon [36].

2.1.1.1 IBM Cell Broadband Engine

The Cell [1,2,50], shown in shown in Figure 2.1, is a heterogeneous chip multi-processor with one general-purpose 64-bit two-way PowerPC SMT core (the Power Processing Element – PPE), and eight vector-processing (128-bit SIMD-RISC) cores (the Synergistic Processing Elements – SPEs), which are specialized for acceleration of data-parallel computations. The on-chip interconnection network of the Cell is a circular ring, termed the Element Interconnect Bus (EIB), which connects all nine cores with memory and an external I/O channel to access other devices, such as the disk and network controller.

The PPE functions as a front-end processor for task scheduling and data distribution between SPEs, as well as for running the operating system. It also provides support to SPEs for executing system calls and services. The SPEs are designed to accelerate data-parallel (vector) computations. The SPEs are specialized processors with software-managed private memories and the programmer is responsible for specifying the data movement between the main

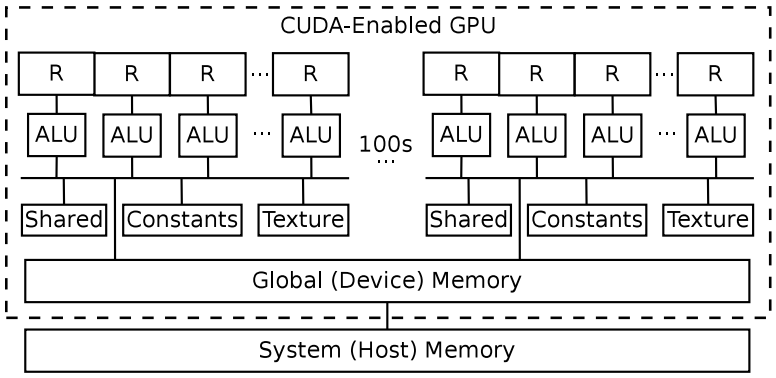


Figure 2.2 CUDA-enabled NVIDIA GPU architecture.

memory and each SPE's local storage using the Cell's coherent DMA mechanism, and can overlap data transfer latency with computation using multiple asynchronous DMAs. This facility allows the programmer to explicitly manage the data flow between Cell components, e.g., for improving I/O performance [45].

2.1.1.2 General-purpose Computing on Graphics Processing Units (GPGPU)

General-purpose computing on graphics processing units (GPGPU) is a common technique used to exploit the parallelism by using the graphics processing unit (GPU) to perform general purpose computing which is traditionally handled by the CPU. Current top-of-the-line GPUs are equipped with dozens of fragment processors and have much higher bandwidth than the traditional CPUs. The processing power of the GPUs have been exploited for multiple scientific [57–63], database [64–68], geometric [69–74] and imaging applications [75–79], and have shown significant performance benefit as compared to traditional CPU and multicore programming. Furthermore, the increase in parallelism within a processor also leads to other desired advantages such a reduced power-consumption [80, 81] and better memory latencies [82–84].

We use CUDA-enabled [4] GPUs in this dissertation. These GPUs are SIMT architectures and provide stream processing capabilities allowing the programmer to execute the parallel portion of the code on GPU devices. Figure 2.2 shows the high-level architecture of a CUDA-enabled NVIDIA GPU. A typical GPU has hundreds of microprocessors group together in a processing blocks having shared and dedicated memory hierarchies. A hardware implemented scheduler efficiently executes a large number of threads at each processing

blocks. CUDA [4] programming framework is generally used to program NVIDIA GPUs. CUDA provides a set of language extensions to the C/C++ programming language to distinguish between GPU executable multithreaded functions and host executable single threaded functions. With improved programming support [85], GPUs are now being introduced in mainstream clusters [36, 86, 87].

2.1.2 MapReduce Programming Model

MapReduce is a parallel programming model for large-scale data processing on parallel and distributed computing systems [31, 88–90]. It provides minimal abstractions, hides architectural details, and supports transparent fault tolerance. The model is a domain-specific, high-productivity alternative to traditional parallel programming languages and libraries for data-intensive computing environments, ranging from enterprise computing [36, 91] to peta-scale scientific computing [31, 90, 92]. Several research activities have engaged in porting MapReduce to multicore architectures [31, 89, 90], whereas recently, leading vendors such as Intel began supporting MapReduce natively in experimental software products [13, 28].

Figure 2.3 shows an example of different MapReduce operations on the Word Count application. The first phase includes the map operations that produces intermediate data in the form of (key, value) pairs. The intermediate data is then grouped together before executing repeated reduce operations to produce the final resultset. The following illustration provides pseudocode for the simple map and reduce operations of the Word Count application shown in Figure 2.3.

```
void map(String document) {
  // document: document contents
  for each word w in document:
    EmitIntermediate(w, '1');
}

void reduce(String word, Iterator partialCounts) {
  // word: a word
  // partialCounts: a list of aggregated partial counts
  int sum = 0;
  for each pc in partialCounts:
    sum += ParseInt(pc);
  Emit(word, AsString(sum));
}
```

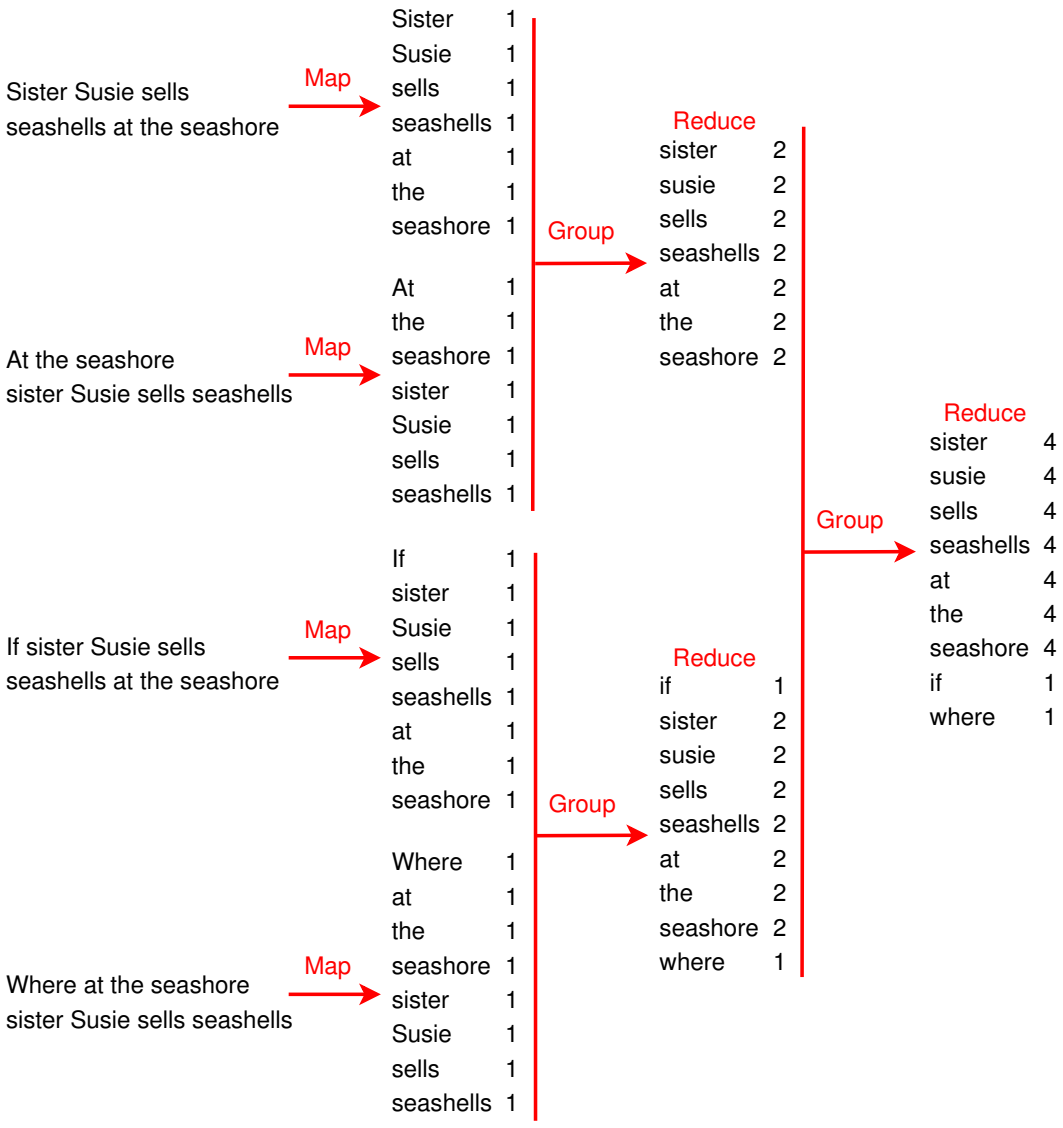


Figure 2.3 Example of MapReduce operations on Word Count application.

MapReduce typically assume homogeneous virtual processors that alternate between mapping, partitioning, sorting, and merging data. While this approach is friendly to the programmer, it adds complexity to the runtime system, if the latter is to manage heterogeneous resources with markedly variable efficiency in executing control-intensive and compute-intensive code. Recent work [93] addresses performance heterogeneity, but is limited only to issues arising from using virtual machines to support compute nodes [36]. Inherent architecture heterogeneity remains a major problem when the cluster components include

specialized accelerators, as in such setups the mapping function should consider individual component capabilities and limitations while scheduling jobs on these resources. Another complication arises because of the assumption that data is distributed between processors before a MapReduce computation begins execution [91]. On distributed systems with accelerators, accelerators use private address spaces that need to be managed explicitly by the runtime system. These private spaces create effectively an additional data distribution and caching layer –due to their typically limited size— which is invisible to programmers but needs to be implemented with the utmost efficiency by the runtime system.

The public implementations of MapReduce for Cell [31] and GPU [89], provides the programmer with a set of APIs for writing MapReduce applications for these architectures. The runtime for Cell implementation divides the execution flow into five stages (Map, Partition, Quick-sort, Merge-sort, and Reduce) and schedules these stages on accelerators cores. This work has shown that compared to standard multicore setups, the Cell can provide performance improvement for computationally intensive workloads with moderate data sets. Similarly, the MapReduce implementation for GPU hides the programming complexity of a GPU behind easy-to-use MapReduce interfaces, and enable the users to write MapReduce applications for GPUs with having to learn about the graphics APIs and GPU architecture.

2.2 Related Work

As previously described in Chapter 1.2, the framework presented in this dissertation focuses on three key areas for efficiently exploiting heterogeneous clusters: easy-to-use programming model, efficient data distribution and prefetching techniques, and adaptive resource management and scheduling techniques, for heterogeneous asymmetric clusters. This section summarizes the prior work that is closely related to these three areas.

2.2.1 Programming Models for Heterogeneous Systems

In this section, we provide an overview of prior work that is closely related to programming asymmetric and heterogeneous systems. We also focus on the programming models and techniques that are used to program accelerator-based systems.

2.2.1.1 Message Passing Interface (MPI)

MPI [33,94,95] is a popular programming model to write parallel applications for distributed systems. The MPI-based programming model adopted in Roadrunner [21] is attuned to its specific hardware. So far, the programmers have relied on manual porting and few automatic tools [21] to run applications on the setup. Running a processor-agnostic programming model, such as MPI, across all cores of any type, implies that accelerators will need to execute the full software stack of communication libraries and task execution control, thus sacrificing precious local storage and execution units that would better be used for dense data-parallel computations. On the plus side, applications that have been "ported" to Roadrunner have shown significant increase in performance [21] compared to that on current state-of-the-art multicore symmetric clusters. This stresses the need for a user-friendly model that will allow applications to easily benefit from such hardware resources.

2.2.1.2 BrookGPU

BrookGPU [96] implements the compiler and runtime environment using stream programming extensions to the C programming language for general purpose computations on a GPU. Each compute kernel in BrookGPU is implemented as a function that is applied to every element in a stream. Data dependencies between compute kernels are identified through explicit declaration of input and output parameters of the stream. StreamIt [97] is a similar programming language and compilation infrastructure that uses the similar streaming approach and applies uniform operations to each kernel elements in multiple input streams to produce a single output stream. Both BrookGPU and StreamIt provide runtimes that map the compute kernels onto processing elements.

2.2.1.3 CUDA

The common practice for programming accelerators, especially GPUs, is to use CUDA [4]. CUDA provides a set of language extensions to the C/C++ programming language to distinguish between GPU executable multithreaded functions and host executable single threaded functions. CUDA also provides a set of APIs to facilitate programmer in allocating GPU memory, copy data between GPU and host memories, and to launch the execution of multithreaded kernel on GPU devices. To facilitate programming, CUDA exposes three special language abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. The use of these abstractions is conducive to the programmer dividing her program

into coarse-grain sub-problems that can be executed independently in parallel. As an additional advantage, individual sub-problems are amenable to be further divided into finer-grain slices, which can also be solved cooperatively in parallel. This arrangement leverages one of the key benefits of threads: enabling them to cooperate with each other while solving individual sub-problems. Finally, the assignment of slices to the physical processors is done by the underlying runtime, enabling flexible parallel designs in which the exact number of physical processors needs not to be known until the runtime.

2.2.1.4 OpenCL

OpenCL [98] provides a development framework for programming heterogeneous platforms comprising of general-purpose processors and GPUs. It provides a C programming language style programming constructs to write compute kernels that can be executed on the attached devices. These compute kernels can also operate on the data structures other than the contiguous streams, and assume Single-Program Multiple Data (SPMD) execution models within the compute kernels. Initially the Both CUDA and OpenCL update local variables in the private memory of the accelerator, enabling the programmer to ignore any side-effect of kernel execution on the main memory. OpenCL was originally supported by a limited vendors, but with its increased popularity almost all major vendors have now started providing its support in their latest accelerator products.

2.2.1.5 Phoenix

Phoenix [90] is a shared-memory implementation of MapReduce for transparently running data-intensive processing tasks on symmetric systems. It provides a thread-based runtime that automatically manages thread creation, dynamic task scheduling, data partitioning, and fault tolerance across symmetric processors. It hides the details of parallel execution from the programmer and requires a functional representation of the algorithm. Phoenix has shown scalable performance for both multicores and conventional symmetric multiprocessors [90]. Although Phoenix can run on a Cell processor, it only utilizes the generic core and does not use the accelerators.

2.2.1.6 Sequoia

Sequoia [32] is a programming language that models heterogeneous systems as trees of memory modules such that the leaves of the tree represent processors. The programmer divides

the program execution as hierarchies of tasks, and maps these hierarchies to the memory subsystems of target machines. Sequoia provides the APIs to facilitate the programmer to describe vertical communication in the tree. It provides a complete programming system including the compiler and runtime framework for the Cell-based processor. Although, Sequoia provides platform-specific programming model for program Cell-based accelerators, it does not provide a generic programming framework to program accelerator-based asymmetric heterogeneous systems.

2.2.1.7 Merge

Merge [28] is a general purpose programming framework for heterogeneous multicore systems similar to that of Sequoia, and uses MapReduce based tree style programming model. Data movements between the tree nodes are orchestrated automatically without the programmer's intervention. It addresses the system heterogeneity by allowing the programmer to specify multiple implementations of each compute kernel for each heterogeneous core in the system. The Merge runtime uses simple sampling scheme to identify the optimal core to execute a particular kernel. Merge provides a solution to program heterogeneous architectures on a single system, however, it does not address the heterogeneity of accelerator-based systems in large-scale distributed settings.

2.2.1.8 Qilin and Harmony

Qilin [99] and Harmony [100] provide execution models for heterogeneous multiprocessors using high-level language constructs and explicit input and output parameters. Programmer can specify the compute kernels in Qilin in Intel Threading Building Blocks (TBB) [101] for CPU executable kernels or NVIDIA CUDA [102] for GPU executable kernels. Both Qilin and Harmony use adaptive mapping to automatically map the compute kernels to the available accelerators at the runtime. These approaches also use analytical performance models to determine the optimal execution time of each compute kernel on available accelerators.

2.2.1.9 Mapping CUDA to Heterogeneous CPUs

The growing acceptance of CUDA has attracted a growing body of researcher to work on mapping the CUDA to other non-GPU based architectures. These works include MCUDA [103], OpenMP to GPU [104], MPI to CUDA [105], and Hadoop using CUDA [89]. All of these approaches except MCUDA map the domain specific languages to GPU, whereas MCUDA

provides a mapping from CUDA to general purpose CPUs. This concept of mapping the CUDA to other architectures has also been exploited in the generation of synthesizable Register Transfer Level (RTL) for FPGAs [106]. Several scalable solutions for programming multicores and many-core architectures for explicitly parallel, BSP [107] programming models have been proposed including Rigel [108], IRAM [109], RAW [110], and Trips [111]. However, CUDA and OpenCL remains the only industry-accepted and widely used programming models to program the heterogeneous accelerators.

2.2.1.10 Limitations of Prior Work

Although, the existing research efforts presented in the section is widely used in their respective settings, none of these efforts addresses the problem of deploying and programming the accelerator-based systems in heterogeneous distributed settings. The prior work presented in this section are either too specific to an installation, or consider some specific assumptions. For example, the control flow representation of the program, input/output data and kernel specification assumptions and requirements in Qilin and Harmony may not hold valid in the accelerator-based asymmetric clusters. The most widely used and scalable programming model, MapReduce, assumes homogeneous computational resources, and hence distributes the workload statically among the available compute nodes. This assumption may not hold valid in accelerator-based setups where different accelerator resource has distinct computational capabilities and memory subsystems. Furthermore, MPI based programming models require low-level knowledge of the underlying architecture requiring expert understanding of the deployed accelerators, and hence are not scalable to variety of accelerator-resources. The tools and technologies presented above, such as BrookGPU, CUDA, OpenCL, and Sequoia, consider some specific accelerators and cannot be generalized on the widely available commodity accelerators such as Cell, GPUs, FPGAs etc.

2.2.2 Data Distribution and Prefetching Techniques

The I/O performance of traditional HPC systems is typically improved through pre-staging of necessary data on computing resources, and node-level OS optimizations. A number of works [112–122] have shown the benefits of staging in reducing data-transfer times and improving HPC system serviceability. At the node level, a mature body of knowledge, comprising simple to advanced pattern-based approaches, exists for data caching [123–141] and prefetching [135, 142–156] to improve I/O performance and bridge the gap between the CPU

and disk access speeds. However, these approaches primarily focus on the I/O performance for traditional systems that do not have limited I/O capabilities, and cannot be simply applied to heterogeneous clusters where limited on-chip memory is available at each accelerator. Moreover, adapting and extending these I/O-improving techniques for heterogeneous clusters will provide opportunities for designing resource orchestration policies that best suit the needs of the target applications. Several related work address the problem of optimally workload distribution and load balancing on asymmetric clusters [157–168].

2.2.2.1 Global Distributed Cache

A recent effort [169] explores an approach of using additional dedicated nodes as global caching nodes to improve the I/O performance in a distributed setting. This approach implements an MPI-IO [170] layer where I/O related tasks on the output data are delegated to the dedicated nodes. This approach assumes MPI programming paradigm, which may not always be used to program accelerator-based heterogeneous systems. Furthermore, since this work is specially designed for applications that produce large output data, its performance impact is reduced to a limited set of HPC applications.

2.2.2.2 Compiler Optimization

A compiler optimization-based approach [171] to manage the GPGPU memory hierarchies and parallelism takes the GPU kernel function as input, analyzes the code, and identifies its memory access pattern. It then generates the updated kernel with the required memory operations and with the kernel invocation parameters. It exploits the memory coalescing and vectorization, if supported by the GPGPU architecture, to optimize the kernel performance. This work targets the applications where static transformation can be performed to optimize the memory references.

2.2.2.3 DataStager

DataStager [172] provides data staging services for large-scale applications for homogeneous clusters. It uses dedicated I/O nodes to move the data from the compute nodes to the storage devices. This approach results in reduced I/O overhead on the application and reduces its processing time. Although this approach is interesting, it does not address the data prefetching challenges of limited on-chip memory at specialized accelerators and coprocessors. Furthermore, DataStager specifically targets output data of the applications, and does

not provide any services to stage input data, which in the case of HPC systems can easily exceed to the order of terabytes.

2.2.2.4 G-Streamline

G-Streamline [173] proposes several heuristic based algorithms and techniques to remove the dynamic irregularities in the memory references and control flows from the input GPU kernels. It resolves these irregularities on the fly by analyzing the interactions between the control and memory operations and their relations with the global program data. Although G-Streamline presents a promising solution that does not require offline profiling, it is only limited to the specific memory hierarchies of GPUs, and applies only to the predictable dynamic data irregularities.

2.2.2.5 *hi*CUDA

*hi*CUDA [174] provides a low-level programming method to translate C-style programs into CUDA. It provides directive-based interfaces to specify the memory and computation operations on the parallel data, and uses source-to-source compilation to generate CUDA compatible code from *hi*CUDA code. The directives provided by *hi*CUDA are unstructured, and require the application programmer to specify all the data transfer operations. This approach is limited to the GPU architectures, and does not eliminate the need for the application programmer to thoroughly understand the GPU memory hierarchies.

2.2.2.6 COMPASS

COMPASS [175] provides a shader-assisted prefetching mechanism that uses idle GPU to prefetch the data for single-threaded CPU applications. It emulates the hardware-based prefetcher and improves the performance of single-threaded application which cannot execute a dedicated prefetching thread. It uses a baseline GPU-architecture and introduce several extensions to it for designing the desired prefetcher in a simulated environment. The applicability of COMPASS is limited by the fact that it is a simulated hardware and it is not provided by any vendor.

2.2.2.7 Overlapping I/O and Computation

Overlapping I/O with computation is a common optimization technique, and has shown significant improvements in the overall system performance [176, 177]. Different computation

and I/O overlapping techniques for MPI-based applications have been studied and proposed in [178]. These efforts present different I/O optimization techniques by overlapping computation with I/O, however, these approaches are applicable to a limited accelerators and coprocessors that support asynchronous I/O.

2.2.2.8 Limitations of Prior Work

The prior research in the field of data distribution and prefetching techniques provide solutions that primarily applicable to homogeneous setups with conventional multicore processors. They cannot be applied directly to heterogeneous setups comprising accelerators and coprocessors. Furthermore, these efforts provide solutions that are applicable to specific installations and application characteristics. Accelerators and coprocessors generally have limited on-chip memory, and cannot implement an I/O optimization stack. Furthermore, they require a general-purpose processor or host machine to initiate the I/O operations. Some of the efforts presented in this section target distributed systems, but require external data staging nodes to perform the prefetching and data handling operations. Such assumptions and requirements cannot stand valid in an accelerator-based heterogeneous system, where accelerators cannot access the network interconnects directly and can only communicate with the host processors.

2.2.3 Resource Management and Scheduling in Heterogeneous Clusters

Heterogeneous resource scheduling has been studied for distributed and grid systems, mostly addressing issues that arise from performance asymmetry instead of architectural heterogeneity. Most of these efforts addressing heterogeneity at the node-level [179–182] and distributed system-level [183–186] represent scientific kernels as graphs with nodes representing interdependent tasks of the entire job. Mesos [187] is a substrate for sharing cluster resources across multiple frameworks, such as Hadoop [91] and MPI [33], and uses two-level scheduling where it first offers resources to a framework, and the framework selects some of the offered resources and uses its own scheduling policy. This approach enables the resource sharing at the granularity of frameworks, but does not incorporate application performance requirements such as deadlines and response times in the presence of load spikes. AJAS [188] provides an adaptive job allocation strategy for heterogeneous clusters, but does not consider varying application load and response time to make decisions.

2.2.3.1 Static Workload Scheduling

A static workload distribution and load-balancing scheme for PC-based heterogeneous clusters has been proposed in [161] that takes into account the computational power of each processor at each node. The nodes having more computational power are assigned bigger tasks than then one with lesser computational resources. This work addresses the heterogeneity of having processors clocked at different rates with different Intel-based ISAs. The approach proposed in this work assumes prior knowledge of the processing power of each node is already available, and uses this knowledge for static assignment of tasks to the cluster nodes. The prior knowledge of the processing power of each node is determined by executing computational and memory intensive benchmarks at each node and then comparing the results to determine the relative computational performance of each node.

2.2.3.2 Dynamic Workload Scheduling

A multi-agent framework for workload scheduling and load balancing is proposed in [168] that first splits the processes into separate jobs and then assign these small jobs into the available cluster nodes in a balanced fashion by using the mobile agents. The processing load on a particular node is determined by the length of the job queue that reflects the total number of unprocessed processes on that node. This work makes use of the mobile agent [189] technology and addresses the issue of heterogeneity in the mobile nodes by simulating the mobile nodes using PMADE (Platform for Mobile Agent Distribution and Execution) [190] on cluster of Windows NT based PCs.

PeraSoft [158] presents a distributed data allocation and sorting algorithm with automatic load-balancing on commercial off-the-shelf Sun workstations running the Linux operating systems. It uses Beowulf parallel-processing architecture from NASA that links commodity off-the-shelf processors to build high-performance cluster. PeraSoft exploits local knowledge in workload distribution such that global processes are completed using the local knowledge and recovery resources. It combines the workload distribution and load balancing while the data is being sorted for processing. Since PeraSoft is designed using Beowulf cluster that does not efficiently support I/O intensive applications, PeraSoft is also limited to be used in compute intensive applications.

A dynamic scheduling scheme for utilizing the spare capabilities of heterogeneous clusters of computers has been presented in [167]. This work is very particular to a scenario where

periodic parallel real-time jobs are already been scheduled on a heterogeneous cluster, and new aperiodic parallel real-time jobs are requested to be executed on the clusters. The cluster scheduler schedules new jobs on the cluster nodes by modeling the spare capabilities of individual nodes. If a real-time job cannot be scheduled such that its completion deadline is met by using the spare capabilities of the nodes while running the periodic real-time jobs, its admission is rejected and is reported back to the user. It uses simple modeling technique to compute the spare capabilities of each node by determining how many resources (processors, memory etc.) are free at any particular time. Each node of the clusters hosts a local scheduler, which operates on Early Deadline First (EDF) policy to schedule periodic as well as aperiodic tasks. Hosting a scheduler at each node in the accelerator-based systems may not work at all, or would impose extra overhead at each node. Furthermore, this approach also requires that the scheduler to know the time required to execute each job on each of the cluster node.

A workload distribution technique for non-dedicated heterogeneous cluster (running generic as well as dedicated tasks) is proposed in [162]. This work also assumes that the heterogeneous cluster is already loaded by dedicated tasks, and new generic tasks are assigned to the cluster. The solution presented in this work uses a queuing model for three different queuing disciplines, namely, dedicated application without priorities, prioritized dedicated applications without preemption, and prioritized dedicated applications with preemption, and schedules the workload between the cluster nodes based on these queuing disciplines such that the overall average response time of the generic applications is minimal.

A task assignment algorithm for heterogeneous computing systems that exploits best-first search technique (A^* algorithm) commonly used in the disciplines of artificial intelligence is presented in [159]. Although, the solution proposed in this work provides an optimal task assignment scheme, it is not suitable to be used in the large-scale scheduling problems because of its high response time and space complexity. The approach used in this work assumes that the assigned tasks can be subdivided and can be represented as an arbitrary task graph with arbitrary costs on the nodes and edges of the graph. The corresponding task graph is then mapped to the cluster nodes using well-defined assignment algorithms to solve the scheduling problem.

2.2.3.3 Fair Scheduling in Homogeneous Clusters

Fair scheduling policies for homogeneous clusters involve allocating each job a fair share of the resources. For instance, if a job takes time t to execute all by itself, then in the presence of n jobs, it should take time nt. Many proposals offer modifications to fair scheduling. Deadline Fair Scheduling [191] provides processes with proportionate-fair CPU time in multiprocessor servers. Delay Scheduling [192] provides a cluster-level fair scheduling scheme that exploits data locality for MapReduce [88] and Dryad [193], but their context does not cover processor heterogeneity or varying application load. A time-sharing based fair scheduling mechanism is presented in [194], which is developed for DryadLINQ [195] cluster. Quincy [196] provides a fair scheduling scheme that preserves and leverages data locality for homogeneous clusters under MapReduce, Hadoop, and Dryad where static application data is stored on the computing nodes. All of these efforts target homogeneous clusters in specific settings, and do not consider varying application load and response times to make scheduling decisions. In a sense, our contribution presented later in this dissertation could be seen as a modification of fair scheduling taking into account load spikes and resource heterogeneity.

2.2.3.4 Limitations of Prior Work

The workload distribution and task scheduling techniques presented in this section are either specific to application characteristics, or does not take into account the computational capabilities and memory capacities of each compute node while distributing the workload and scheduling the computational tasks. Furthermore, these techniques are generally designed for general-purpose PC-based clusters, and are either intended for homogeneous clusters or address the cluster heterogeneity only at the operating system and clock-speed level. Some of the approaches presented in this section require assumptions and scenarios, such as having prior knowledge of the application completion time at each node, a specific task arrival pattern at the cluster manager, or non data-intensive tasks assignments to the cluster. None of these assumptions can hold valid in scientific and enterprise settings where accelerator-based heterogeneous compute nodes are deployed as workhorses in distributed settings to satisfy the computing needs of the clusters.

2.3 Chapter Summary

In this chapter, we have discussed the details of enabling technologies and the related work on programming models, data distribution, and resource management techniques for heterogeneous clusters. Our framework presented in this dissertation aims to provide adaptive and scalable programming models and efficient resource management techniques for heterogeneous clusters, with the main objective of utilizing the heterogeneous cluster resources in a transparent and efficient manner.

Chapter 3

Scalable Programming Framework for Heterogeneous Clusters

Asymmetric parallel architectures are rapidly being established in emerging systems as the *sine qua non* for achieving high performance without compromising reliability. The model has been realized in asymmetric multi-core processors, where a fixed transistor budget is heavily invested on many simple, tightly coupled, accelerator-type cores. These cores provide custom features that enable acceleration of computational kernels operating on vector data. Accelerator cores are controlled by relatively few conventional processor cores, which also run system services and manage off-chip communication. Researchers have collected mounting evidence on the superiority of asymmetric multi-core processors in terms of performance, scalability, and power-efficiency [8, 10, 12]. The advent of the Cell processor and GPUs as HPC and data processing engines [15–21, 45], further attests to the potential of asymmetric architectures.

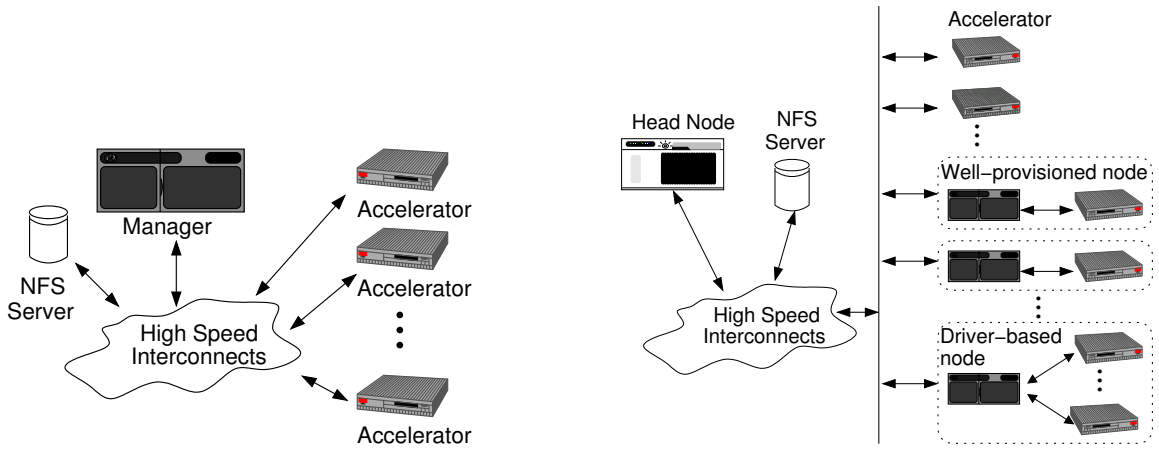
While parallel programming models for symmetric clusters have been studied at length, the synthesis of parallel programming models for asymmetric parallel architectures is an open problem. In particular, hiding the architectural asymmetry from the programming model, and exploiting the vast computational density of accelerators while they communicate with inherently slower system components remain major challenges. One good candidate from homogeneous parallel programming model that can be adapted for asymmetric HPC clusters is MapReduce [88]. MapReduce is a simple model for machine-independent parallel programming at large scales. It provides minimal abstractions, hides architectural details including heterogeneity, and supports transparent fault tolerance. While current implementations of MapReduce accommodate standalone accelerators, e.g., Cell [31], and take into consideration heterogeneity of compute nodes due to virtualization in the task scheduler [93],

they do not cope with the asymmetry between the computational density of accelerators and data processing and forwarding capabilities of manager nodes. This asymmetry can lead to severe performance penalties by exposing communication or I/O bottlenecks.

In this chapter, we propose two approaches that can be used to efficiently program accelerator-based heterogeneous clusters. First we present CellMR, a MapReduce-based extended programming framework for asymmetric HPC clusters with large-memory general purpose head nodes and accelerator-type compute nodes. CellMR hides asymmetry and enables high-performance, cost-effective, and scalable data processing. We target HPC clusters with heterogeneous processor architectures, similar to LANL's RoadRunner [21], built however with low-cost compute nodes that capitalize on the compute density of graphics and gaming processors. While CellMR can be extended to arbitrary hybrid parallel architectures, we first evaluate our efforts on a cluster that uses the Cell, arguably one of the dominant asymmetric multi-core processors, as an accelerator, and then we extend CellMR to different resource configurations including hierarchical setups. CellMR uses data streaming approach to effectively support MapReduce computations, and strives to overlap completely I/O and communication latencies. CellMR supersedes data transfer and task management libraries for asymmetric accelerator-based architectures, such as IBM's ALF [40], which delegate parameterization and optimization of scheduling data transfers to the application developers. CellMR transparently adapts the parameters of data streaming and task scheduling to the application at runtime, thereby relieving developers of some significant programming effort. CellMR also removes I/O bottlenecks via use of techniques such as asynchronous accesses and double-buffering at multiple levels of the system. Second, we explore an approach to lowering the barrier to entry for users and organizations that need to take advantage of the computing power and energy efficiency offered by accelerator-based clusters. Our approach leverages the advances in component-based software engineering, in which complex software systems are constructed from reusable and adaptable components. In particular, we explore how layered software architectures can alleviate many of the difficulties of building and maintaining component-based systems on accelerator-based clusters.

3.1 Asymmetric System Architecture

Figure 3.1 illustrates a high-level view of heterogeneous system with symmetric (Figure 3.1(a)) as well as asymmetric (Figure 3.1(b)) compute nodes. The manager and all the compute nodes are connected via a high-speed network, e.g., Gigabit Ethernet. Application



(a) Symmetric compute nodes.

(b) Asymmetric compute nodes.

Figure 3.1 High-level system architecture of symmetric and asymmetric clusters.

data is hosted on a distributed file system, such as the Network File System (NFS) [197] or Lustre [198]. In our implementation, we used NFS for baseline comparative investigation of our alternate approaches. The server manages a number of back-end accelerator-based nodes and is responsible for scheduling jobs, distributing data, allocating work between compute nodes, and providing other support services as the front-end of the cluster. The actual data processing load is carried by the Cell-based accelerators.

A typical MapReduce setup consists of a dedicated front-end machine that handles job scheduling and resource management for a number of back-end resources. Similarly as in typical symmetric clusters, the front-end node is a general purpose multi-core server with a large amount of DRAM, which acts as a cluster *manager*. The difference is that, in CellMR, the back-end compute nodes are asymmetric cell-based accelerators, PS3s, instead of generic computers. The manager distributes and schedules the workload to the compute nodes. The generic core on the compute nodes then uses MapReduce to map its assigned workload to the accelerator cores. In essence, the programming model of CellMR resembles a two-level MapReduce: the front-end maps workloads to the cell-based back-ends, and the back-end generic core maps the workload to its accelerators.

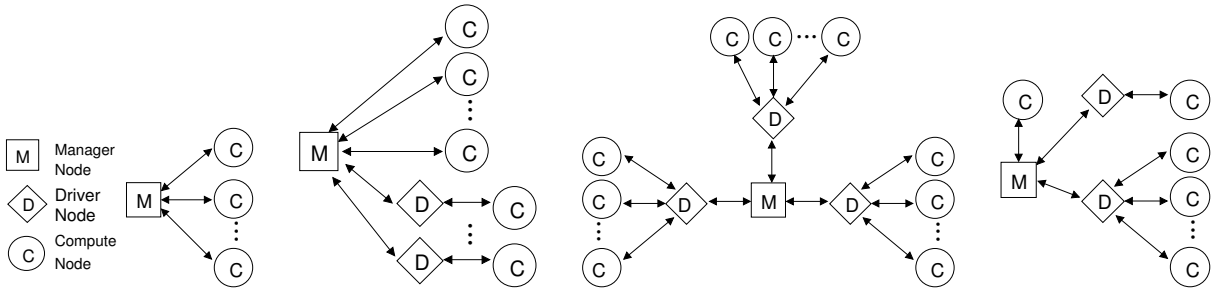
3.1.1 Targeted Accelerators

Addressing the inherent imbalances of accelerator-based asymmetric clusters while hiding the associated complexity from users is key to achieving high performance and high productivity. Although designing for all possible resource configurations and types of accelerators is very complicated, we characterize the accelerator-based resources that we plan to use in this dissertation based on the general-purpose computing and system management capabilities of the accelerators. More specifically, we consider three classes of accelerators:

Self-managed well-provisioned accelerators These accelerators have high compute density, along with on-chip capabilities to efficiently run control code and self-manage I/O and communication. For example, an accelerator coupled with several general-purpose processor cores on the same chip falls into this category. The on-chip computational power of the general-purpose cores and the amount of memory attached to the accelerators is assumed to be sufficient for self-management, in the sense that the control code running for scheduling tasks and performing communication on the general-purpose cores does not become the major performance bottleneck. Asymmetric multi-cores, such as IBM Cell [1] processors fall under this category.

Resource-constrained well-provisioned accelerators These accelerators have high compute density but insufficient on-chip general-purpose computing capability for running control code and/or insufficient on-board memory for self-managing I/O and communication. I/O and communication are managed by an external, dedicated, node with general-purpose cores, which acts as a *driver* for the accelerators. Programmable FPGAs or GPGPUs, such as CUDA-enabled NVIDIA and OpenCL-enabled AMD/ATI GPUs fall under this category. A conventional host processor is required in these settings to run the operating system and provide general-purpose I/O and communication capabilities to the accelerators, which communicate with the host over the I/O bus.

Resource-constrained shared-driver accelerators These accelerators are similar to the previous case, however drivers are shared among several accelerators, to yield a potentially more cost-efficient design. These accelerators are similar to the previous case, however drivers are shared among several accelerators, to yield a potentially more cost-efficient design. Advanced multi-chip programmable GPUs, such as NVIDIA GeForce GTX 295 [199], or installations with multiple programmable accelerators or FPGAs on a single driver node



(a) *Conf I:* Self-managed well-provisioned accelerators.

(b) *Conf II:* Resource-constrained well-provisioned accelerators.

(c) *Conf III:* Resource-constrained shared-driver accelerators.

(d) *Conf IV:* Mixed accelerators.

Figure 3.2 Resource configurations for enabling asymmetric clusters.

fall under this category. Note that the driver for these accelerators can itself support heterogeneous accelerators within a single compute node. This driver organization in effect creates an additional level of asymmetry in the system.

3.1.2 Resource Configurations

We consider four resource configurations for the target asymmetric clusters as shown in Figure 3.2. The configurations are driven by the type of the back-end components used, as well as by economical constraints and performance goals. In all cases, the manager and all back-end nodes are connected via a high-speed commodity network, e.g., Gigabit Ethernet. Application data is hosted on a distributed file system (NFS [197] in our implementation).

The first configuration (Figure 3.2(a)) we consider is that of self-managed well-provisioned accelerators (*Conf I*), connected directly to the manager. A blade with Cell processors [200] including multi-Gigabyte DRAM and high-speed network connectivity would fall into this category. Small-scale academic settings may also adopt such a configuration, using, e.g., PS3 nodes and scaling down the workload per PS3 so as to not exceed the limited DRAM capacity and not stress the limited general-purpose processing capabilities of the PS3. The compute nodes execute directly all MapReduce tasks and the manager merges partial results from the computes nodes.

The next configuration (Figure 3.2(b)) uses resource-constrained well-provisioned accelerators (*Conf II*). Each driver provides large memory space, communication and I/O capabilities to an individual resource-constrained accelerator, e.g. a PS3. The manager distributes in-

put data to the driver nodes in large chunks. The driver nodes proceed by streaming these chunks to the attached accelerators. Accelerators execute the MapReduce tasks, however, partial results produced by accelerators are merged at the corresponding driver nodes and the manager executes the global merge operation on the results received from the driver nodes.

The use of a single driver per resource-constrained accelerator is not always justifiable as one accelerator may not be able to fully utilize the driver's resources. In contrast, a single manager may not be sufficient to match the data demands of many accelerators simultaneously. We address this by using a hierarchical setup (*Conf III*), so that each driver node manages multiple accelerator nodes (Figure 3.2(c)).

The use of a single driver per resource-constrained accelerator is not always justifiable as one accelerator may not be able to fully utilize the driver's resources. In contrast, a single manager may not be sufficient to match the data demands of many accelerators simultaneously. We address this by using a hierarchical setup (*Conf III*), so that each driver node manages multiple accelerator nodes (Figure 3.2(c)).

Finally, an asymmetric system may employ a mix of the above configurations based on particular requirements. We capture this mix in our last configuration (*Conf IV*) (Figure 3.2(d)). In this case, the manager is agnostic of the class of the attached compute nodes and simply divides the input workload between available compute nodes. The execution of MapReduce tasks and merging of partial results are managed automatically at each component, while the final result is produced by the manager, which performs the global merge of the results received from the attached drivers.

3.2 MapReduce-based Extended Programming Framework

We use MapReduce as the cluster programming model. The applications developed using MapReduce process data in parallel using two simple primitives: A *map* primitive that maps an input of (key, value) pairs to an output of intermediate (key, value) pairs; and a *reduce* primitive, that merges the values associated with each key. The runtime system partitions the output of the map stage between nodes and sorts the input to each node before applying the reduction. MapReduce implementations provide a runtime library for expressing

data-intensive parallel computation using the *map* and *reduce* primitives. This programming model has a high-enough level of abstraction to hide many of the complexities of parallel programming, such as partitioning, mapping, load balancing and tolerating faults, while providing adequate capabilities that can be exploited for managing heterogeneous resources in the runtime system.

The manager divides the MapReduce tasks (map, reduce, and sort etc.) in small workloads, and assign these workloads to the attached accelerator-based nodes. Irrespective of the type of back-end nodes, the manager transparently distributes and schedules the workload to them. If the back-end is a self-managed accelerator, its general-purpose core uses MapReduce to map the assigned workload to the accelerator cores (SPEs). In contrast, if the back-end is driver-based, the driver components further distribute the assigned workload to the attached accelerator node(s). Note that the manager differs from a driver. Drivers execute control tasks for communication and I/O on behalf of accelerators, whereas the manager controls work and data distribution for the entire cluster. This model can be thought of as a hierarchical MapReduce: each level maps the workload to the next level of nodes, until it reaches the compute node, i.e., the Cell processor, where the generic on-chip core maps the workload to the accelerators.

3.2.1 Extending MapReduce for Heterogeneous Resources

In a typical MapReduce setting, Map and Reduce tasks are scheduled separately on potentially distinct sets of cluster nodes. In our enhanced MapReduce runtime, a data segment is assigned to a compute node and the entire sequence of MapReduce operations on that data segment is executed on that compute node. This way, our implementation does not require classifying cluster resources as *mappers* or *reducers*; the data segment stays on the assigned node and both the operations are performed on it at that node, thus providing improved locality. One of the disadvantages of having separate mappers and reducers is that the reducers cannot start the reduce process before the completion of all mappers. In our MapReduce runtime, the manager does not wait for all nodes to complete their processing before a global merge operation is executed. Instead, the manager starts to merge the results as soon as results are received from more than one compute nodes.

3.2.2 Programming Asymmetric Clusters

From an application programmer's point of view, irrespectively of the resource configuration employed, MapReduce is used on asymmetric resources as follows. The application is divided into three parts.

1. The code to initialize the runtime environment. This corresponds to the time spent in a MapReduce application but outside of the actual MapReduce data processing stages and includes initialization, data distribution and finalization. This part is unique to our design and does not have a corresponding operation in prior MapReduce implementations.

2. The code that runs on the accelerator cores and does the actual data processing for the application. This is similar to a standard MapReduce application setup running on a small portion of the input data that has been assigned to the compute node. It includes both a map phase to distribute the workload between the accelerator cores, and a reduce phase to merge the data produced from accelerators.

3. The code that runs on the manager to merge partial results from each compute node into a complete result. This code is invoked every time a result is received from a compute node and executes a global merge phase that is identical in operation to the reduce phase on each compute node.

All map, reduce, and merge functions are application-specific and should be provided by the programmer. Once identified, the binaries for the above components are generated for all the available targets (different accelerators and conventional multicore processors) in the system. The availability of these binaries enables our system to transparently schedule tasks at any time, on any type of accelerator and hide heterogeneity and asymmetry. Furthermore, it frees the programmer from system level details such as managing the memory subsystems of the accelerators, orchestrating data transfers between the manager and the compute nodes, and implementing optimized communication mechanisms between cluster nodes, and enables the programmer to focus on the application-specific part of code.

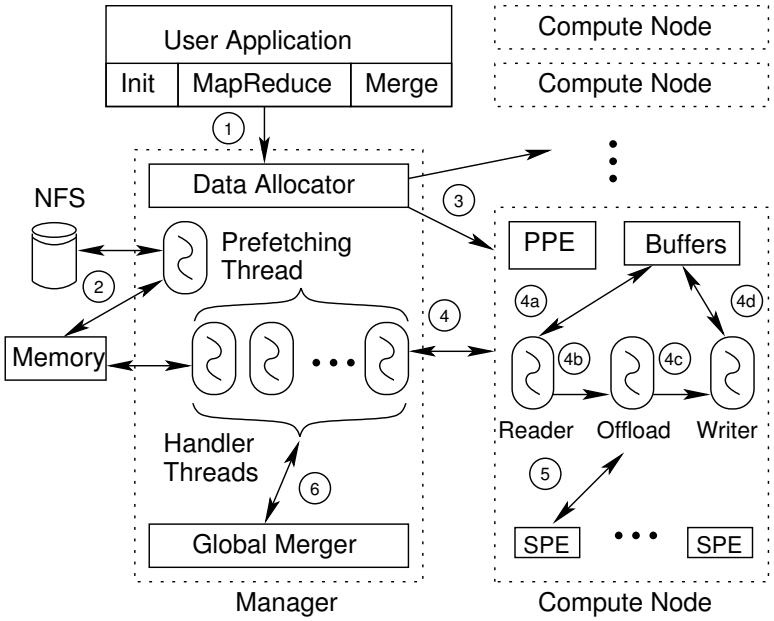


Figure 3.3 Interactions between CellMR components.

3.2.3 Data Management Operations

In this section, we describe the runtime interactions between the various software components at the manager and each of the compute nodes, as depicted in Figure 3.3.

3.2.3.1 Manager Operations

The manager handles job scheduling, data hosting, and data distribution among drivers or compute nodes. We use well-established standard techniques for the first two tasks and focus on compute-node management and data distribution in this discussion. Once an application begins execution (Step 1 in Figure 3.3), the manager loads a portion of the associated input data from the file system (NFS in our current implementation) into its memory (Step 2). This is done to ensure that sufficient data is readily available for compute nodes, and to avoid any I/O bottleneck that can hinder performance. For well-provisioned compute-nodes with drivers, this step is replaced by direct prefetching on the drivers.

Next, client tasks are started on the available compute nodes (Step 3). These tasks essentially self-schedule their work by requesting input data from the manager, processing it, and returning the results back to the manager in a continuous loop (Step 4). For well-provisioned nodes, the result data is directly written to the file system, and the manager is informed

of task completion only. Once the manager receives the results, it merges them (Step 6) to produce the final result set for the application. When all the in-memory loaded data has been processed by the clients, the manager loads another portion of the input data into memory (Step 2), and the whole process continues until the entire input has been consumed. This model is similar to using a large number of small map operations in standard MapReduce.

3.2.3.2 Compute Node Operations

Application tasks are invoked on the compute nodes (Step 3), and begin to execute a request, process, and reply (Steps 4a to 4d) loop. We refer to the amount of application data processed in a single iteration on a compute node as a *work unit*. With the exception of an application-specific *Offload function*[1] to perform computations on the incoming data, our framework on the compute nodes provides all other functionality, including communication with the manager (or driver) and preparing data buffers for input and output. Each compute node has three main threads that operate on multiple buffers for working on and transferring data to/from the manager or disk. One thread (Reader) is responsible for requesting and receiving new data from the manager (Step 4a). The data is placed in a receiving buffer. When data has been received, the receiving buffer is handed over to an Offload thread (Step 4b), and the Reader thread then requests more data until all available receiving buffers have been utilized. The Offload thread invokes the Offload function (Step 5) on the accelerator cores with a pointer to the receiving buffer, the data type of the work unit (specified by the User Application on the manager node), and size of the work unit. Since the input buffer passed to the Offload function is also its output buffer, all these parameters are read-write parameters. This is to give the Offload function abilities to resize the buffer, change the data type, and change the data size depending on the application. When the Offload function completes, the recent output buffer is handed over to a Writer thread (Step 4c), which returns the results back to the manager and releases the buffer for reuse by the Reader thread (Step 4d). Note that the compute node supports variable size work units, and can dynamically adjust the size of buffers at runtime.

The driver in our resource configurations interacts with the accelerator node similarly as the manager interacts with the compute nodes. The difference between the manager and the driver node is that the manager may have to interact and stream data to multiple compute

[1]The function that processes each work unit on the accelerator-type cores of the compute node. The result from the Offload function is merged by the GPP PowerPC core on the Cell to produce the output data that is returned to the manager.

nodes, while the driver only manages a single accelerator node. The driver further splits the input data received from the manager and passes it to the compute node in optimal size chunks as discussed in the following section.

3.2.4 Evaluation

We evaluate CellMR using eight Sony PS3 compute nodes connected via 1 Gbps Ethernet to a manager node. The manager has two quad-core Intel Xeon 3 GHz processors, 16 GB main memory, 650 GB hard disk, and runs Linux Fedora Core 8. The manager also runs an NFS server. Of the 8 SPEs of the Cell, only 6 SPEs are visible to the programmer [45, 201] on the PS3.

3.2.4.1 Applications

We use the following well-known MapReduce applications to study the effect of the various design alternatives for the symmetric as well as asymmetric heterogeneous cluster. These applications originate from scientific computing environments, including epidemiology, environmental science, image segmentation, and statistical analysis [202–204]. For our evaluation, we used four common MapReduce applications. These applications are classified based on the MapReduce phase where they spend most of the execution time. A brief description of the applications that we have ported to our framework is provided below.

- *Linear Regression:* This application takes as input a large set of 2 dimensional points, and determines a linear best fit for the given points. This is a map-dominated application.
- *Word Count:* This application counts the frequency of each unique word in a given input file. The output is a list of unique words found in the input along with their corresponding occurrence counts. This is a partition-dominated application.
- *Histogram:* This application takes as input a bitmap image, and produces the frequency count of each RGB color composition in the image. This is a partition-dominated application.
- *K-Means:* This application takes a set of points in an N-dimensional space and groups them into a predefined number of clusters with approximately equal number of points in each cluster. This is a partition-dominated application.

Input (MB)	Linear Regression	Word Count	Histogram	K-Means
4	0.34	1.95	1.06	1.66
64	2.88	501.76	45.66	167.93
128	12.56	-	318.66	-
192	21.81	-	394.78	-
256	34.89	-	-	-

Table 3.1 Execution time (sec.) on stand-alone PS3.

Table 3.1 shows the average execution time for running the four benchmarks on a stand-alone accelerator without using our framework. Note that Linear Regression is the only benchmark that successfully completes for all input sizes. All other benchmarks incur swapping and run out of swap space with smaller input sizes. Also note the rapid growth in the completion time due to excessive swapping as the input size is increased.

3.2.4.2 Evaluation with Symmetric Heterogeneous Cluster

We now present the evaluation of CellMR with a symmetric cluster using eight Sony PS3 compute nodes connected via 1 Gbps Ethernet to a manager node. The manager has two quad-core Intel Xeon 3 GHz processors, 16 GB main memory, 650 GB hard disk, and runs Linux Fedora Core 8. The manager also runs an NFS server. Of the 8 SPEs of the Cell, only 6 SPEs are visible to the programmer [45, 201] on the PS3. Each PS3 node has a swap space of 512 MB and runs Linux Fedora Core 7. We evaluate the performance of CellMR in a symmetric heterogenous cluster shown in Figure 3.1(a) using Cell-based PS3 nodes as compute nodes. For the experiments, we use the following resource configurations:

- *Single* configuration, which runs the benchmarks on a stand alone PS3, with data provided from an NFS server to factor out any effects of the PS3's slow local disk. *Single* provides a measure of performance of one small-memory, high-performance computational accelerator running the benchmarks.
- *Basic* configuration uses the manager and compute nodes as follows. The manager equally divides the input at the beginning of the job and assigns it to the compute nodes in one step. The manager then waits for the data to be processed, before merging individual output to produce the final results. *Basic* serves as the baseline for evaluating streaming and dynamic work unit scaling in CellMR.

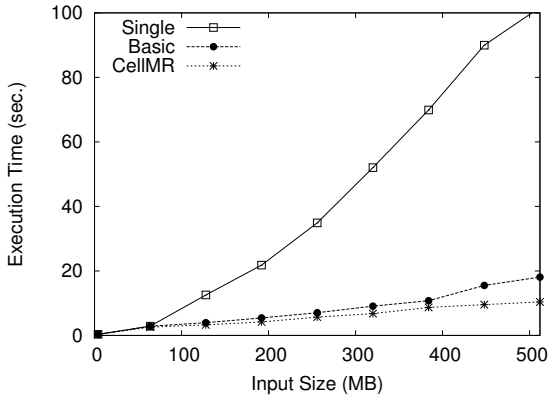


Figure 3.4 Linear Regression execution time with increasing input size on symmetric cluster.

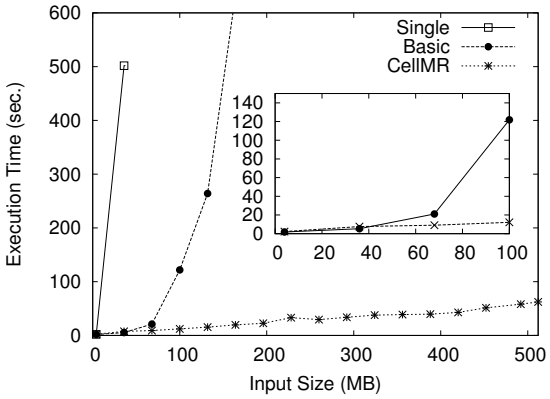


Figure 3.5 Word Count execution time with increasing input size on symmetric cluster.

- *CellMR* configuration that also uses all nodes but employs CellMR for work unit scaling and scheduling.

Benchmark Performance In the following, we describe the performance of Linear Regression, Word Count, Histogram and K-Means benchmarks using the above-mentioned resource configurations.

Linear Regression For this benchmark, we chose input sizes ranging from 2^{22} points (file size 4 MB) to 2^{29} points (512 MB). Figure 3.4 shows the results. Under *Single*, the input data quickly becomes larger than the available physical memory, resulting in increased swapping, and consequently increases the execution time. In *Basic*, a smaller fraction of the data is sent to each of the PS3s, which relieves the memory pressure on them somewhat. Initially, CellMR performs slightly better than *Basic*, 18.9% on average for input size less than 400 MB, mostly due to its data streaming characteristics and work unit size optimizations. However, as the input size is increased beyond 400 MB, the peak virtual memory footprint for *Basic* is observed to grow over 338 MB, much greater than the 200 MB available memory, leading to increased swapping. Once *Basic* starts to swap, its execution time increases noticeably. In contrast, CellMR is able to dynamically adjust the work unit size to avoid swapping on the PS3s, thus, achieving 24.3% average speedup across all the considered input sizes compared to *Basic*.

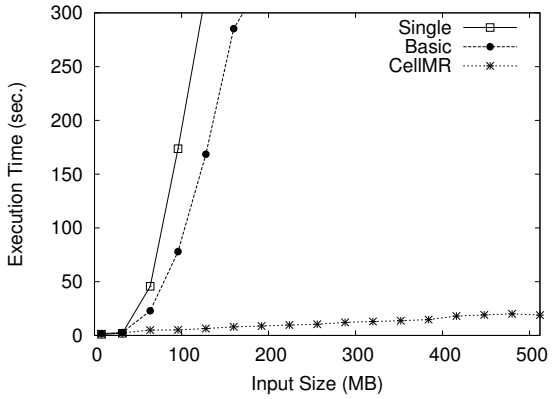


Figure 3.6 Histogram execution time with increasing input size on symmetric cluster.

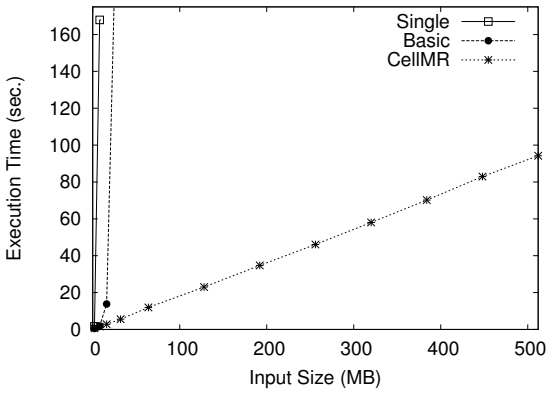


Figure 3.7 K-Means execution time with increasing input size on symmetric cluster.

Word Count During our experiments with Word Count, we observed exponential growth in memory consumption relative to the input data size, since each input word would emit additional intermediate data out of the map function. Therefore, for any input size greater than 44 MB, *Single* experienced excessive thrashing that caused the PS3 node to run out of available swap space (512 MB) and ultimately crash. Similarly, *Basic* was also unable to handle input data sizes greater than 176 MB, and took 631.9 seconds for an input size of 164 MB. Figure 3.5 shows the results. Here, CellMR is not only able to process any input size, it outperforms *Basic* by 65.3% on average for the points, emphasized in the inset in the figure, where *Basic* completed without thrashing (input data size < 96 MB).

Histogram Figure 3.6 shows the result for running Histogram under the three test configurations. It can be observed that CellMR scales linearly with the input data size. In contrast, *Basic* only scales initially, but then looses performance as the increased input size triggers swapping, e.g., for an input size of 160 MB, the peak virtual memory size grows to 285 MB and it takes 285.3 seconds to complete. On average across all input points less than 192 MB, CellMR does 68.2% better than *Basic* for Histogram. The maximum input size that *Single* and *Basic* can handle without crashing is 192 MB, for which the execution times are 394.8 and 328.6 seconds, respectively.

K-Means The results for the K-Means benchmark are shown in Figure 3.7. Note that K-Means use a different number of iterations for different input sizes. Therefore, considering total execution times for different inputs does not provide a fair comparison of the effect of

Configuration	# of Drivers	# of PS3s	PS3s per Driver
Conf I	-	8	-
Conf II	8	8	1
Conf III	2	8	4
Conf IV	5	8	4,1

Table 3.2 Resource distribution under different configurations.

increasing input size. We remedy this by reporting the execution time per iteration in the figure. While the CellMR implementation scales linearly, *Single* and *Basic* use up all the available virtual memory with relatively low input sizes. Both *Single* and *Basic* crash for an input size greater than 8 MB and 32 MB, respectively. For 32 MB input size, *Basic* takes over 319 seconds/iteration compared to 5.5 seconds/iteration of CellMR. The only input size where the *Basic* case doesn't thrash is for 1 MB. In this instance *Basic* outperforms CellMR by 17%, as this input size is too small to amortize the management function overhead of CellMR. However, this is not of concern, as with any input size greater than 1 MB, CellMR does significantly better than *Basic*.

3.2.4.3 Evaluation with Asymmetric Heterogeneous Clusters

We now present the evaluation of CellMR with asymmetric clusters in various configurations as shown in Figure 3.2. All resource configurations use CellMR to execute the assigned workload.

Our testbed consists of eight Sony PS3s, a manager node, and an 8-node x86 multi-core cluster, where each node can serve as a driver. All components are connected via 1 Gbps Ethernet. The manage is the same as described in Section 3.2.4.2.

Table 3.2 shows the distribution of resources that we use for each of the configurations presented in Figure 3.2, in addition to the manager node. Note that in *Conf I*, the PS3s are connected directly to the manager, and in *Conf IV*, four PS3s share a driver, while each of the other four has a dedicated driver. Moreover, in all the test configurations, the total number of accelerators is fixed, i.e., 8 PS3s or 48 SPEs and only the resource arrangement is varied.

Benchmark Performance We now examine the effect of different resource configurations on the average execution time for each of our benchmarks.

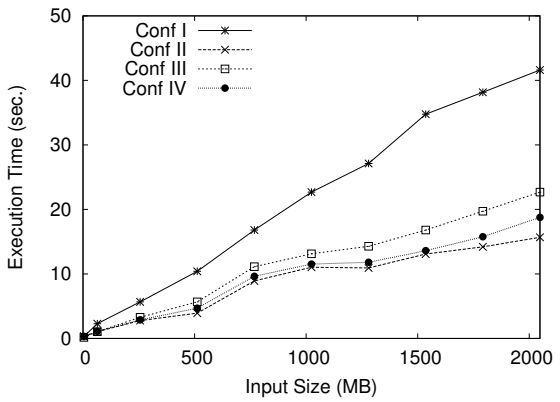


Figure 3.8 Linear Regression execution time with increasing input size on asymmetric cluster.

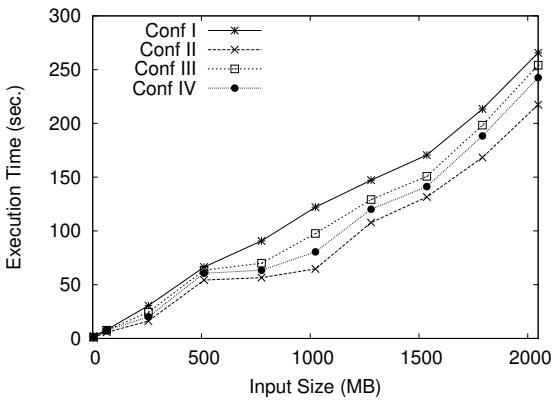


Figure 3.9 Word Count execution time with increasing input size on asymmetric cluster.

Linear Regression For this benchmark, the input size ranges from 2^{22} points (4 MB) to 2^{31} points (2 GB). Figure 3.8 shows the average execution time for running the Linear Regression benchmark with increasing input size under different resource configurations. All four resource configurations show similar scaling patterns with the increasing input size. Overall *Conf II* performs 53.1% better than *Conf I*, since each driver node in *Conf II* makes use of its large memory to store and process the intermediate results from the attached PS3s. For similar reasons, i.e., having a higher number of drivers to handle the PS3s, *Conf II* performs 14.4% and 8.0% better than *Conf III* and *Conf IV*, respectively.

Word Count For Word Count, we observe an exponential growth in memory consumption relative to the input data size, since each word emits additional intermediate data out of the map function. This has the direct impact on the execution time as shown in Table 3.1. For any input size greater than 44 MB, a single accelerator node thrashes and runs out of available swap space (512 MB). However, all the resource configurations in our setup are not only able to process any input size, but also complete the benchmark without thrashing, with linear increase in execution time with increasing input size (Figure 3.9). Once again, *Conf II* outperforms *Conf I*, *Conf III* and *Conf IV* by 32.5%, 19.2% and 12.7%, respectively, since job scheduling and merging tasks are distributed efficiently between driver nodes.

Histogram Figure 3.10 shows the average execution time for running the Histogram benchmark under the four test configurations. On average *Conf II* performs 25.1%, 17.8% and 11.1% better than *Conf I*, *Conf III* and *Conf IV*, respectively. In our experiment with a

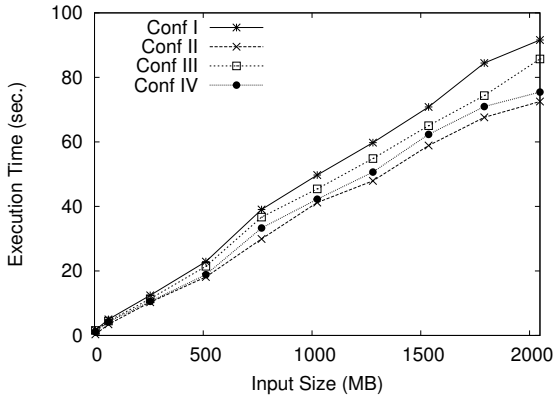


Figure 3.10 Histogram execution time with increasing input size on asymmetric cluster.

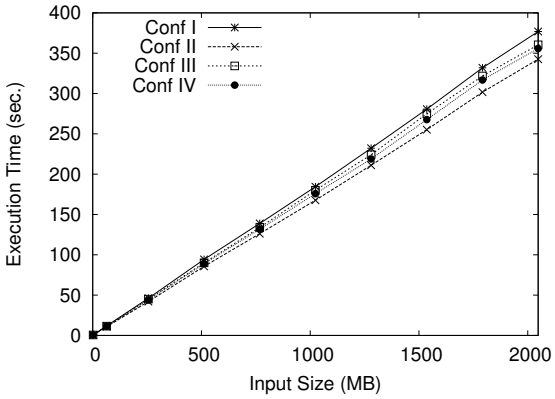


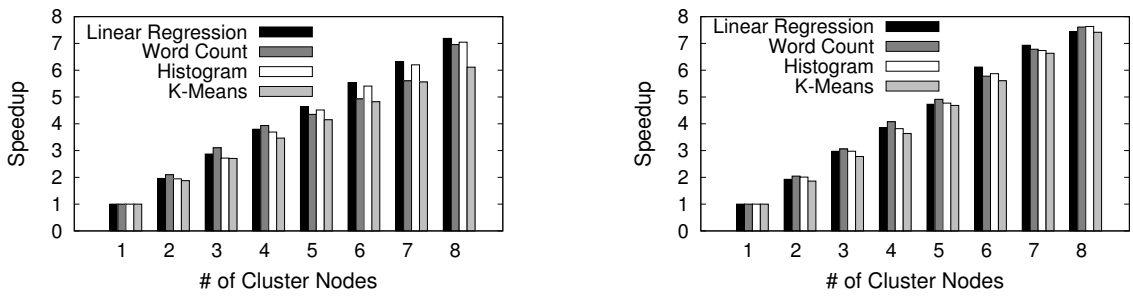
Figure 3.11 K-Means execution time with increasing input size on asymmetric cluster.

stand-alone PS3, we observe that the execution time for 192 MB input size is 394.8 seconds because of excessive swapping of intermediate data. This benchmark also shows that our design scales linearly for any input size for all tested configurations.

K-Means Figure 3.11 shows the results for the K-Means benchmark. K-Means uses a different number of iterations for different input sizes. Therefore, considering total execution times for different inputs does not provide a fair comparison of the effect of increasing input size. We remedy this by reporting the execution time per iteration in the figure. The result for this benchmark shows that *Conf II* outperforms *Conf I*, *Conf III* and *Conf IV* by 9.0%, 6.6% and 4.4% respectively. Just as in the case of previous benchmarks, the improvement comes from the fact that in *Conf II* each accelerator node has more memory and computation resources in the form of a dedicated front-end node attached with it.

3.2.4.4 Scaling Characteristics

We observe how the performance of our benchmarks scale with the number of accelerator nodes using *Conf I*, *Conf II*, and *Conf III*. Figure 3.12(a) shows the speedup in performance normalized to the case of 1 node in *Conf I* and *Conf II*. Both these configurations have similar speedups because of similar manager to compute node relationship, and are shown in a single graph. For *Conf III*, we only have enough PS3s to scale up to using four drivers with four PS3s each. However, we emulate up to 8 drivers as follows. During our tests with 1 to 4 drivers, we observe near identical load on the manager from each of the drivers. Based on



(a) *Conf I/II.* (b) *Conf III.*

Figure 3.12 Effect of scaling on resource configurations.

this observation, we create a test-loader that generates the same requests to the manager as that of a driver with accelerators and use it to scale the experiment beyond four accelerators.

Figure 3.12(b) shows the speedup for our benchmarks in *Conf III*. We use the same input size for all runs of an application. However, the input sizes for the different applications are chosen to be large enough to benefit from using 8 nodes: 512 MB for Linear Regression and Histogram, 200 MB for Word Count, and 128 MB for K-Means. The curve of K-Means is based on time per iteration, as explained earlier.

Although we are only able to evaluate scaling on the relatively modest scale of 8 nodes, our results show that our framework scales almost linearly as the number of compute nodes increases and this behavior persists for all the benchmark. However, we observe that the improvement trend does not hold for all benchmarks in *Conf I/II* when the eighth node is added. Upon further investigation, we find that the network bandwidth utilization for such cases is quite high, as much as 107 MB/s compared to the maximum observed value of 111 MB/s on our network, measured using remote copy of a large file. High network utilization introduces communication delays even with double buffering and prevents our framework from achieving a linear speedup. However, if the ratio of time spent in computation compared to that in communication is high, which is the case in scientific applications, we can obtain near linear speedup. We test this hypothesis by artificially increasing our compute time for Linear Regression by a factor of 10, which results in a speedup of 7.8. For *Conf III*, no such network bottleneck exists, since each driver manages the attached accelerator using a dedicated connection.

3.3 Advanced Software Engineering Approach to Program Heterogeneous Clusters

The impressive performance advantages provided by heterogeneous accelerator-based clusters make them desirable computing facilities for researchers and enterprises alike [21, 36]. The heterogeneity of the hardware units of accelerator-based cluster, the complexity of their API, and the ad-hoc nature of their communication interfaces confine the use of these powerful and power-efficient computing facilities to all but advanced computer users. Researchers in other fields — whose simulations and computer-models for investigating a myriad of fields, e.g., medicine, high-speed physics, etc., can benefit from such resources — are simply left out. The current state-of-the-art is such that one literally needs to be a seasoned computer scientist to be simply able to set up and use an accelerator-based heterogeneous cluster, let alone optimize and derive peak performance from one.

Layered architectures are a proven approach for expressing the logic of complex computer systems [205–208], with several implementation techniques. Some of these techniques require specialized languages or language extensions. We explore an approach called *mixin-layers* [209], which provides all the benefits of layered architectures within the confines of standard C++ [210]. This design choice is influenced by the unique requirements of our target environment.

A typical accelerator-based cluster requires coordination of the execution of multiple heterogeneous devices connected to each other through a high-speed interconnect. Due to the ubiquity of C++, one can find a standards-compliant C++ compiler almost on any computing device and operating system. One of the advantages of C++ is its natural interoperability with C. Thus, even if a C++ compiler is not available on some esoteric device, it is always possible to write a module in C and link it with the encompassing C++ component.

In this section, we present the design and implementation of a reusable and adaptable software component framework for setting up accelerator-based heterogeneous systems. The framework provides both ease-of-use and extensibility advantages. We started with our optimized implementation described in the previous sections concerned with optimizing accelerator-based heterogeneous clusters. As is commonly the case, our initial point was an hand-coded implementation for particular deployment environment, with the resulting code not easily reusable or adaptable. We present the results of rearchitecting this initial

code into a mixin-layer-based implementation that provides reusable and adaptable software components and reduce the application deployment time for heterogeneous clusters.

3.3.1 Feature-Oriented Programming and Mixin-Layers

Feature-oriented programming (FOP) is a software development methodology in which features are first-class citizens in the software architecture [211, 212]. FOP decomposes applications into a set of features that together provide the requisite functionality. Composing multiple objects from a single set of features separates the core functionality of an object from its refinements making the resulting software more reusable and robust.

In addition, FOP allows easy mix-and-match composition of features in a modular fashion, as an application is built using step-wise refinement. A common implementation strategy in FOP is to use a layered architecture, in which layers correspond to features. The resulting "feature stack" is composed of many layers with each layer (1) providing a single feature, and (2) refining existing features in the stack [213].

To incrementally refine and flexibly compose features of an accelerator-based cluster cluster, we use mixin-layers, a novel layered architectures' implementation that uses advanced C++ programming techniques. Mixin-layers model different collaborating roles within each layer by means of inner classes [209]. Inner classes mirror the inheritance relationships of their enclosing classes in the layer above. With mixin-layers, the programmer can add functionality flexibly but systematically: each added layer supplies those inner classes that provide the required functionality.

Mixin-layers fits our design goals: it implements features as collaborations of smaller, encapsulated units, each of which can be refined step-wise. As our experimental infrastructure matures, new research issues may warrant switching to another implementation strategy that better satisfies the newly-discovered set of requirements. For example, domain-specific languages have been shown to be effective for flexibly composing features [214–216]. Following a different implementation strategy, however, would still leverage the key conceptual contributions of our approach: demonstrating how a reusable and adaptable software component framework can streamline the process of composing accelerator-based heterogeneous clusters.

3.3.2 Software Components for Heterogeneous Clusters

A major challenge in designing a component-based system is to decide what functionality should be included into which software components. Among the major criteria to be considered is the intuitiveness of the client interfaces and ease-of-reuse. Other criteria tend to be more domain-specific, defined by the constraints of a given computing platform. In this case, our major objective is to encapsulate reusable functionality that can serve as convenient building blocks for constructing accelerator-based clusters. We aim at hiding much of the low-level implementation details from those programmers who simply want to use these software components to quickly put together a cluster of accelerator engines. In the following discussion, we outline the main software components that we chose to make available as part of our infrastructure. We support our decision to expose this particular set of functionality as software components by describing their functionality and client interfaces.

The software components are divided between manager and compute node roles. Figure 3.13 shows different manager and compute nodes software components that provide the execution logic of our design, and their corresponding interactions with each other. In this following, we describe how the functionality of accelerator-based cluster can be decomposed into distinct software components and explain the functionality of these components. These software components are reusable and can be maintained with minimal effort. Furthermore, the components can be used to design heterogeneous accelerator-based clusters comprising different types of accelerators. In the following we explain the functionality of these components.

3.3.2.1 Manager Components

We now describe the main components of our manager layer.

Communicator The *Communicator* component encapsulates the methods required to communicate between different heterogeneous resources of the cluster. This component also implements communication and computation overlapping opportunities by implementing techniques such as double-buffering for each end of the communicating device. The *Communicator* is an extensible module, which can also be used to introduce hardware-component-specific optimizations such as hiding communication latency and improving bandwidth utilization.

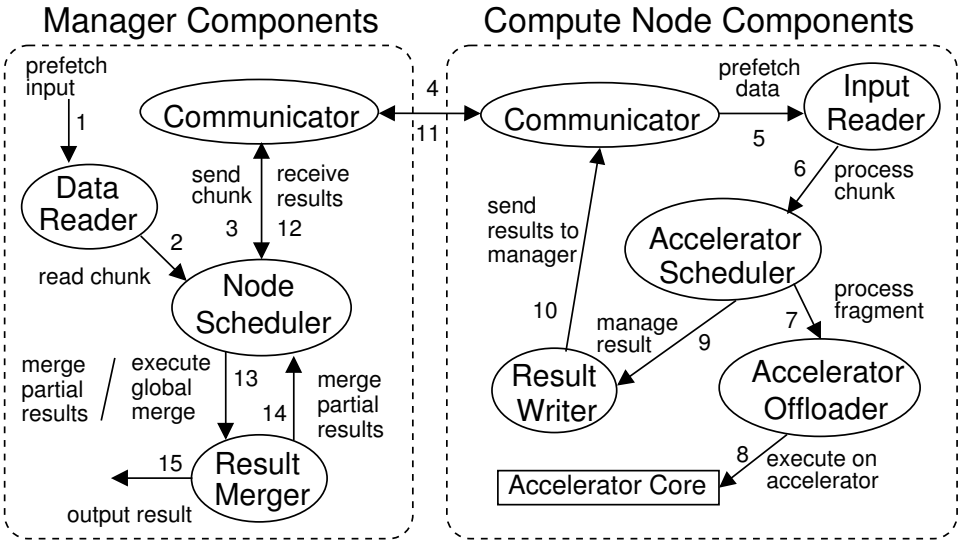


Figure 3.13 Manager and compute nodes components and their interactions.

Data Reader This component prefetches the user specified input data and stores it in the framework according to the application requirement. The data is read in large chunks from the storage device — to amortize access costs such as disk seek times — and further divided and arranged based on application-desired layout. The newly arranged data is then passed to the *Node Scheduler* for streaming to available compute nodes. The *Data Reader* provides interfaces to manipulate the prefetched data and to change the layout as desired.

Node Scheduler The *Node Scheduler* component manages a compute node. For each compute node available to the cluster, the manager initiates one instance of the *Node Scheduler*. Each *Node Scheduler* receives a chunk of unprocessed data from the *Data Reader*, and further optimally streams it to the corresponding accelerator-based compute node. The streaming is attuned to the compute node's data handling capabilities, and can be specified by the user in this component. The streaming process is continued until the entire input data is consumed and processed by the compute node.

The *Node Scheduler* also retrieves the results from the compute nodes, and sends the partial-results to the compute nodes for final merging, as explained next.

Result Merger The *Result Merger* exposes the interfaces to the programmer for including application-specific mechanisms for merging partial results from individual compute nodes into a combined result set, and the global merging criteria for producing the final results

at the manager node. The *Node Scheduler* retrieves and passes on the partial results from compute nodes to the *Result Merger* as the results become available. The *Result Merger* combines the partial results based on the application-specific criteria, e.g., in-order sort, and perform the specified global-merge function to produce the final results. Note that, if needed, the *Result Merger* may also use the *Node Scheduler* again for offloading the combining and merging operations to the accelerator-based compute nodes to improve performance.

3.3.2.2 Compute Node Components

We now describe the main components of the compute nodes layer as shown in Figure 3.13, and how data is manipulated between the different components at the compute nodes.

Input Reader The *Input Reader* at the compute node prefetches blocks of data from the manager, and passes it to the *Accelerator Scheduler* component. We have used multiple buffers for reading the data from the manager to overlap the communication and computational latency, as well as provided support for specifying more optimizations as necessary. If an empty buffer is available, the *Input Reader* initiates a request for another block of data. If no empty buffers are available for the input data, *Input Reader* simply waits until some buffers become free.

Accelerator Scheduler The *Accelerator Scheduler* component schedules the data read from the *Input Reader* for execution on the attached computational accelerator of the node. This component is specific to the kind of attached accelerator of the node, and provides the opportunity to implement any device specific optimization. The input buffer from the *Input Reader* is further divided into small slices suitable to fit into the available memory of the corresponding accelerator. These small slices of data are then scheduled to be streamed to the *Accelerator Offloader* component for execution on the accelerator.

Accelerator Offloader The *Accelerator Offloader* component provides an interface to execute the device specific offload routines, such as application specific data processing and merging functions. This component provides abstractions for the methods required to integrate device-specific programming languages, such as C for CUDA, and C for PS3, with a general-purpose language such as C++. The application and accelerator specific compute routines can be specified in separate files to maintain the modularity of the framework.

The *Accelerator Offloader* reads the input data from the *Accelerator Scheduler* and processes

it on the attached *Accelerator Core*. The *Accelerator Core* represents the specific accelerator device, such as Cell or GPU, which executes application specific task on the given data and produces the result. The results are then returned back to the *Accelerator Scheduler* that passes them to the *Result Writer* component.

Result Writer The *Result Writer* component receives the results from the *Accelerator Scheduler*, and sends them to the manager. Once a block of result data is sent to the manager, its associated buffer is marked as free and can be reused by the *Input Reader*. Note that the result from the *Accelerator Offloader* can not be sent directly to the *Result Writer* component without involving the *Accelerator Scheduler* component. The *Accelerator Scheduler* component should be notified when the processing of a particular data slice is completed at the *Accelerator Offloader* component so that the next data slice can be scheduled. If results are sent directly to the *Result Writer* component without involving the *Accelerator Scheduler* component, then some signaling mechanism needs to be implemented to notify the *Accelerator Scheduler* component that the processing of a particular data slice has been completed, which would increase the synchronization overhead between components.

3.3.3 Illustrative Example

We now present an illustrative example of a real cluster to describe how different components of our design operate and interact with each other. By describing each component's functionality, this example illustrates how our framework orchestrates pre-existing software components to complete a concrete computational task. In particular, we focus on Word Count application discussed in the earlier sections, and show how the computationally-intensive problem of counting word frequencies in text files can be naturally decomposed for efficient execution on an accelerator-based cluster.

The heterogeneous cluster that we used in this example consists of two PS3 compute nodes, two GPU-based compute nodes, and a manager, all connected via a high-speed network.

3.3.3.1 Synthesizing the Application

First, we discuss the software components needed for the Word Count application. Programming-wise, the entire functionality of each cluster node is encapsulated in a template instantiation of the required components for the node. For our example application, the template instantiation for the manager node is as follows:

```
typedef Manager<WordCount> manager;

#define NUM_PS3             2
#define NUM_GPU             2

manager::DataReader                 datReader;
manager::NodeScheduler<PS3>         PS3Schedule[NUM_PS3];
manager::NodeScheduler<GPU>         GPUSchedule[NUM_GPU];
manager::ResultMerger               resMerger;

void main () {
  datReader.startReadingThread();
  for (int i = 0; i < NUM_PS3; ++i) {
    PS3Schedule[i].startSchedulingThread();
  }
  for (int j = 0; j < NUM_GPU; ++j) {
    GPUSchedule[j].startSchedulingThread();
  }
  resMerger.doMerging();
}
```

The corresponding template instantiation for a PS3 compute node is as follows:

```
typedef PS3<ComputeNode<WordCount>> cNode;

cNode::InputReader                  inpReader;
cNode::AcceleratorScheduler         accSchedule;
cNode::AcceleratorOffloader         accOffloader;
cNode::ResultWriter                 resWriter;

void main () {
  inpReader.startReadingThread();
  accSchedule.startSchedulingThread();
  accOffloader.startOffloadingThread();
  resWriter.doWriting();
}
```

Finally, a GPU-based compute node's template instantiation is as follows:

```
typedef GPU<ComputeNode<WordCount>> cNode;

cNode::InputReader                  inpReader;
cNode::AcceleratorScheduler         accSchedule;
cNode::AcceleratorOffloader         accOffloader;
cNode::ResultWriter                 resWriter;
```

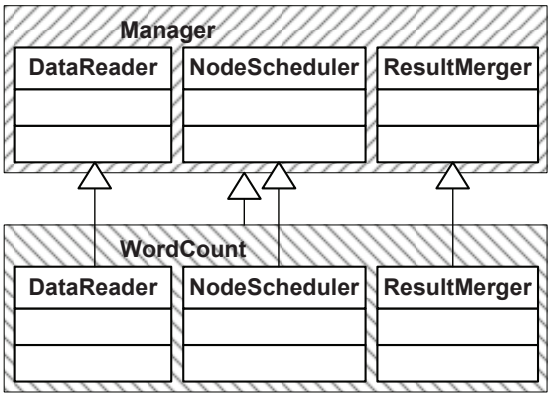
```
void main () {
  inpReader.startReadingThread();
  accSchedule.startSchedulingThread();
  accOffloader.startOffloadingThread();
  resWriter.doWriting();
}
```

Note the functionality of each component as described in Section 3.3.2, e.g., *Input/Data Reader*, is expressed as an inner class in each of the above template instantiations.

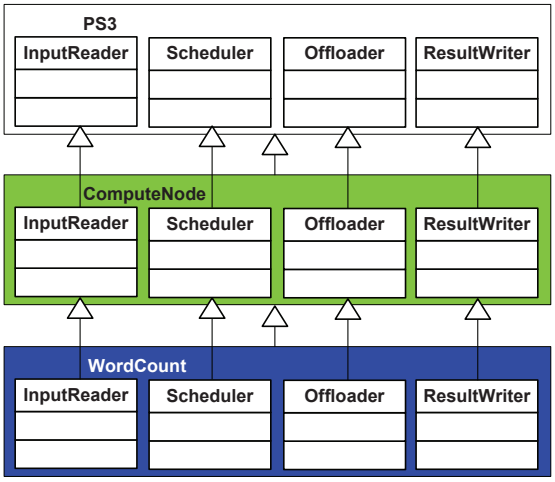
Figure 3.14 illustrates how different software components are represented as mixin-layers, both for the manager node as well as for the Cell and GPU-based compute nodes. Each layer represents a component, defined as a unit of functionality with multiple roles. For example, the `ComputeNode` component defines the `InputReader`, `Scheduler`, `Offloader`, and `ResultWriter` roles. These roles define the distinct operations that are used during the execution of a generic `ComputeNode`. Each layer is added to the composition to either refine or extend the existing components. For example, the `GPU` or `PS3` components add the functionality in their roles that are specific to their respective architectures.

Implementation-wise, each layer is implemented as a template C++ class, whose inner template classes comprise the layer's roles. Both the main component classes and their roles participate in the inheritance relationship with the corresponding classes in the layer above. Thus, to reuse a component, with all its roles, the programmer only has to include that component into a template instantiation. As long as the component has the needed roles (which can be ensured by following careful design practices), its functionality becomes immediately available for constructing any application.

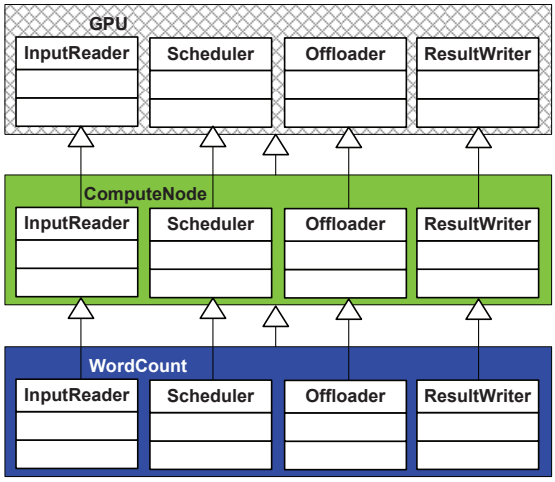
For the Word Count application whose code listings appear above, two out of three components for the compute nodes can be reused out-of-the-box. In the figure, the reused components are colored (shaded) identically. Even though the reusable components will need to be recompiled for different hardware architectures, their functionality will remain the same. This small but realistic example demonstrates how a layered software architecture can be leveraged to provide easy-to-use-and-reuse software components, which can be both architecture independent or device specific. This observation leads us to believe that following this software construction paradigm has the potential to alleviate many implementation complexities for the average programmer.



(a) Mixin-layers for manager node.



(b) Mixin-layers for Cell-based compute node.



(c) Mixin-layers for GPU-based compute node.

Figure 3.14 Manager and compute nodes mixin-layers and the defined roles.

3.3.3.2 Runtime Interactions

Next, we describe how the components we have described above are used at runtime. The execution control flow steps in this discussion are illustrated in Figure 3.13 as numbers along the arrows.

The cluster receives a request to start computing the frequencies of each word in a set of disk files. This causes the *Data Reader* component, located at the manager node, to be invoked (Step 1). The *Data Reader* prefetches and divides the input text file in small chunks. As chunks are read into memory, a separate *Node Scheduler* component for each compute node is instantiated at the manager node, a total of four in this example. Each *Node Scheduler* component reads the next available chunk from the *Data Reader* (2), and divides it into smaller blocks: 4 MB blocks for PS3 nodes, 12 MB blocks for our GPU nodes. Then, the *Communicator* transmits the scheduled data blocks to their target compute nodes (3, 4). This process is repeated until the computation is complete.

Once a compute node is done with counting different word frequencies in its assigned block, the result is sent back (11, 12) to the manager via the node's associated *Result Writer*. At the manager node, the *Result Merger* combines all the received word lists by sorting them as required for Word Count (13). When the *Result Merger* is done with sorting, the combined word lists must be processed for combining repeated word counts to determine final word frequencies. Since the counting of repetitions is computationally intensive, the work must be once again distributed among the compute nodes (14). As before, this distribution task is accomplished by the *Node Scheduler*. The final consolidated result is then computed by the *Result Merger* component after all the compute nodes have finished their computations (13).

On the compute node, the *Input Reader* is responsible for retrieving the assigned blocks from the manager (5). The received blocks are then passed to the *Accelerator Scheduler* (6) in a loop. The *Accelerator Scheduler* is unique to each accelerator engine. Specifically, these components encode the logic required to divide the assigned blocks into slices that can be processed by the underlying architecture — 32 KB for PS3 and 256 KB for our GPUs. Each slice is then passed to the *Accelerator Offloader* component (7), which then either counts the different word frequencies in the assigned slice initially, or counts repeated words in a list for merging repetitions later (8).

Once the *Accelerator Scheduler* has received all the results computed for each data slice, they are passed to the *Result Writer* (9), which in turn sends them back to the manager node by

means of the *Communicator* (10). The results are then reported to the user (15), completing the application run.

Finally, if the hardware configuration is changed, the programmer can easily reuse many of the software components, thus saving time and reducing time-to-solution.

3.3.4 Evaluation

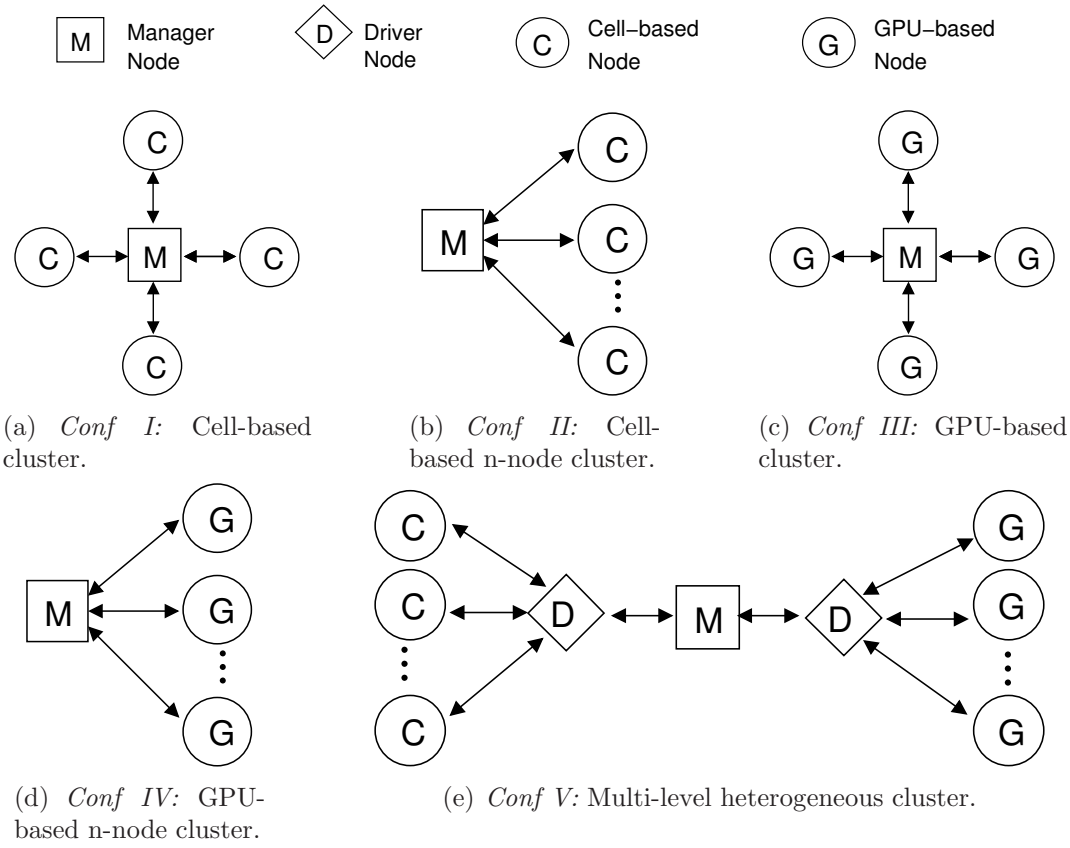
We have implemented a prototype of our design as lightweight libraries for each of the platforms, i.e., x86 on the manager and driver, PowerPC and SPE on the Cell-based PS3 compute nodes, and GPU-based x86 compute nodes using only about 1650 lines of C/C++ and CUDA code. The libraries provide the application programmers with necessary constructs for using different components of the framework.

In our implementation for *Conf V*, we leverage the reusability of the components to build a hierarchical accelerator-based cluster. For instance, the driver node is primarily composed of components reused from the manager and the compute nodes, i.e., the driver instantiates the same `InputReader` as that used for the compute node in the remaining configurations for reading the data from the manager. Moreover, the `NodeScheduler` and `ResultMerger` on the driver are instances of code designed for the manager in the other configurations and is used to manage the attached computational accelerators and merge the partial results. Finally, the `ResultWriter` is similar to that of the compute nodes, and is used to return the results back to the manager.

3.3.4.1 Resource Configurations

Figure 3.15 shows different resource configurations of heterogeneous clusters that we have considered while evaluating our reusable components. Although not exhaustive, we believe these configurations cover most of the cases encountered in accelerator-based cluster design. We have used accelerators of various capabilities as the computing devices (or compute nodes) in different configurations.

Our first configuration, *Conf I* (Figure 3.15(a)), consists of four Cell-based accelerator nodes connected directly with the manager node via high-speed interconnect network (1 Gbps Ethernet in our case). Our second configuration, *Conf II* (Figure 3.15(b)), is a generalization of *Conf I* to n Cell-based accelerators. In these configurations, any workload assigned to the manager is dynamically divided among the attached accelerator nodes by the manager node.



(a) *Conf I:* Cell-based cluster.

(b) *Conf II:* Cell-based n-node cluster.

(c) *Conf III:* GPU-based cluster.

(d) *Conf IV:* GPU-based n-node cluster.

(e) *Conf V:* Multi-level heterogeneous cluster.

Figure 3.15 Resource configurations for Cell and GPU based heterogeneous clusters.

Our third and fourth configurations, i.e. *Conf III* (Figure 3.15(c)) and *Conf IV* (Figure 3.15(d)), are similar to *Conf 1* and *Conf II*, respectively, but instead of Cell-based accelerators, use GPU-based computational accelerators.

The fifth configuration, *Conf V* (Figure 3.15(e)), employs a mix of Cell-based and GPU-based computational accelerators in a hierarchical settings. Both the Cell-based and GPU-based compute nodes are connected with the manager node through a driver node, which acts as a 'local manager' for the attached accelerator nodes. Here, any workload assigned to the manager is dynamically divided between the attached driver nodes that further divide the assigned tasks to the attached computational accelerators based on the accelerators' capabilities. In this configuration, the manager node has to interact with only two driver nodes instead of all the accelerator-based compute nodes of the cluster, thus the driver nodes alleviate from the manager node the pressure of fine-grained computational resource management and scheduling.

3.3.4.2 Experimental Setup

Our testbed consists of several Sony PS3s and GPU enabled Toshiba Qosmio laptop computers, a manager node, and an 2-node standard multicore cluster to serve as drivers. All components are connected via 1 Gbps Ethernet. The manager has two quad-core Intel Xeon 3 GHz processors, 16 GB main memory, 650 GB hard disk, and runs Linux Fedora Core 8. The driver nodes are identical to the manager except that they have 8 GB of main memory. Each GPU enabled Toshiba Qosmio laptop computers has Intel Dual-Core 2 GHz processor, and 4 GB of main memory installed in it. Moreover, each of the laptops has one GeForce 9600M GT [217] CUDA enabled GPU device, with 32 cores and 512 MB of memory, and uses CUDA toolkit 2.2.

For our experiments, we distribute the resources as described in Section 3.3.4.1. For *Conf II* and *Conf IV* we set n=10. In *Conf V* the two drivers have four PS3s and four GPUs connected to them, respectively, and are connected with the manager node. Our goal is to determine the effect of different heterogeneous environments in our prototype implementation, specifically, varying the number, type, and hierarchy of the accelerators.

We conduct the experiments using our prototype implementation that uses the mixin-layers for building high-performance accelerator-based clusters. The focus is on evaluating our design decisions and to investigate how well we can reuse the mixin-based components in different benchmarks applications, and how well it performs as compared to a hand-tuned implementation of the benchmark applications. We have used well-known parallel applications namely Linear Regression, Word Count, Histogram, and K-Means described in Section 3.2.4.

3.3.4.3 Mixin-Layer Components Reusability

We evaluate the effectiveness of our framework in reducing the amount of software-engineering effort for designing applications for the targeted asymmetric hardware resources. One potentially confusing issue is the meaning of the term *component*. In a mixin-layer composition, a component is a template class whose functionality is defined by its inner classes. What is more important for this evaluation is our unit of reusability. Even though the unit of reusability is an entire mixin-layer component, any instantiation can use only the needed roles by simply creating objects of the appropriate inner classes. Therefore, each role can be reused independently of the other roles in the same layer. Therefore, while our implementation is component-based, our measurements are specific to the software engineering metrics

Component (Class)	Linear Regression LOC	Code reuse (in %)		
		Word Count	Histogram	K-Means
`Communicator`	360	100	100	100
`DataReader`	42	14.2	11.9	42.8
`NodeScheduler`	423	100	100	100
`ResultMerger`	38	38.9	40.1	27.1

Table 3.3 Manager classes and corresponding reusable LOC (wrt. Linear Regression) for benchmark applications in *Conf I*.

(e.g., lines of code) of the inner classes.

Code Reusability for Different Applications In this set of experiments, we evaluated how well we can reuse the mixin-layer components across our benchmark applications while keeping the cluster hardware configurations and resources fixed. Table 3.3 shows the total number of lines of codes (LOC) of the major classes at the manager node for Linear Regression along with the percentage of LOC that we were able to reuse across different applications, all in *Conf I*. Note that except for the `DataReader` and `ResultMerger` classes that are unique to different applications, all the other classes, i.e., `NodeScheduler` and `Communicator` are reused by the benchmarks without any modifications. Nonetheless, since we have used modular programming in our implementation, we were able to use some parts of our code in the application-specific user-defined classes as well. Overall 90.7% of the code of the classes at the manager node is reused between different applications in *Conf I* of our implementation.

Table 3.4 shows the total number of LOC of the major classes at the compute nodes for Linear Regression in *Conf I*. Here, except for the `AcceleratorOffloader` class, which contains the application specific routines executed on the computational accelerators of the compute nodes, all the other major classes, i.e., `InputReader`, `AcceleratorScheduler`, and `ResultWriter` are reused without any modifications. Our evaluation shows that overall 64.7% of the code of the classes at compute nodes can be used across different applications. However, the `AcceleratorOffload` class can not be reused as a whole since it is unique for each application, even so by using the mixin-layers abstractions we are able to use 23.6% on average across all benchmark applications.

Code Reusability for Different Configurations In this set of experiments, we fix the application (Linear Regression) and vary the hardware configuration to determine how

Component (Class)	Linear Regression LOC	Code reuse (in %)		
		Word Count	Histogram	K-Means
`InputReader`	92	100	100	100
`AcceleratorScheduler`	78	100	100	100
`AcceleratorOffloader`	148	15.4	19.5	35.8
`ResultWriter`	102	100	100	100

Table 3.4 Compute node classes and corresponding reusable LOC (wrt. Linear Regression) for benchmark applications in *Conf I*.

Component (Class)	*Conf I* LOC	Code reuse (in %)			
		Conf II	*Conf III*	*Conf IV*	*Conf V* wrt. *I* & *III*
`Communicator`	360	100	100	100	100
`DataReader`	42	100	100	100	90.4
`NodeScheduler`	423	100	52.2	52.2	100
`ResultMerger`	38	100	100	100	94.7

Table 3.5 Manager components and corresponding reusable LOC for Linear Regression benchmark in different configurations.

well the mixin-layer components can be reused across different hardware. Table 3.5 shows the total number of LOC of the major classes at the manager node in *Conf I* along with the percentage of the LOC that we were able to reuse for *Conf II*, *Conf III*, *Conf IV*, and *Conf V*, respectively. As observed from the table, we were able to use all of the classes of the manager except the `NodeScheduler` class. This is because `NodeScheduler` interacts with the attached accelerator-based compute nodes that change across the configurations. However, we were still able to reuse 52.2% of our code because of our component-based development approach. Note that in *Conf V*, we have reused 99.3% of the overall code since the driver node is composed of some of the roles of the manager and the compute nodes. The node scheduler component in *Conf V* is exactly the same as the manager node, since the driver node manages the attached nodes in the same fashion as the manager node manages the compute node. The difference in *Conf V* comes in the `DataReader` component, where the driver node reads the data either from the manager, or from a distributed storage. In the case of reading the input from a distributed storage, the driver reads only the part of input that is assigned to it by the manager node. Another difference in the driver comes in `ResultMerger` component, where in addition to merging the results, it sends the merged results to the manager node.

Table 3.6 shows the total number of LOC of the major classes at the compute nodes in *Conf*

Component (Class)	*Conf I* LOC	Code reuse (in %)			
		Conf II	*Conf III*	*Conf IV*	*Conf V*
`InputReader`	92	100	62.5	62.5	98.1
`AcceleratorScheduler`	78	100	71.5	71.5	99.1
`AcceleratorOffloader`	148	100	20.5	20.5	100
`ResultWriter`	102	100	85.2	85.2	98.6

Table 3.6 Compute node components and corresponding reusable LOC for Linear Regression benchmark in different configurations.

I for Linear Regression benchmark, and the LOC that we were able to reuse across different hardware configurations. Here, we have reused 100% of our code while moving from *Conf I* to *Conf II*, and also from *Conf III* to *Conf IV*, as the only difference here is the number of compute nodes. Interestingly, we reused 54.9% of our overall compute node code while transforming our code from *Conf I* to *Conf III*. This is because different type of accelerators offer different optimization opportunities for the corresponding type of attached accelerator, and we have to modify our accelerator code base for the new accelerator to exploit the specific optimization opportunities available. Note that most of the source code that has to be modified for supporting a new accelerator is in the platform specific `AcceleratorOffloader` class, which as discussed earlier, contains the application-specific functions that are executed on the accelerator of the compute node. Also, note that while reporting the code reusability for *Conf V*, we compare it with the *Conf I* and *Conf III* since the compute nodes in *Conf V* are the combination of the two configurations.

3.3.4.4 Benchmark Performance

In this set of experiments, we focus on the performance of the mixin-layers based code generated for our studied accelerator-based cluster configurations. First, we compare our code with available hand-tuned implementations presented in Section 3.2.4 of the benchmarks. We have extended our hand-tuned implementations from Cell-based compute nodes to GPU-based compute nodes. We have implemented several optimizations in our hand-tuned implementations. These optimizations include implementing locality-aware data distribution between compute nodes, implementing optimal offloading workload size for Cell and GPU based nodes, and implementing double buffering at each communication layer (manager-to-driver and driver-to-compute node) to overlap computation with communication.

Table 3.7 shows the comparison of the execution time for both implementations using *Conf V*.

Application	Execution Time (sec.)		Overhead (%)
	Hand-Tuned	Mixin-Layer	
Linear Regression	10.1	10.4	2.0
Word Count	65.3	66.4	1.5
Histogram	22.1	22.9	1.8
K-Means	93.4	94.2	0.8

Table 3.7 Execution time and overhead for hand-tuned and mixin-layer implementations for benchmark applications using *Conf V*.

Application	Execution Time (sec.)				
	Conf I	***Conf II***	***Conf III***	***Conf IV***	***Conf V***
Linear Regression	10.4	4.7	39.8	16.5	18.0
Word Count	66.4	28.2	238.8	97.2	110.3
Histogram	22.9	11.0	86.8	36.5	39.8
K-Means	94.2	50.5	329.4	144.5	159.4

Table 3.8 Benchmark execution time with different resource configurations.

Note that the hand-tuned implementation is unique for each benchmark application, and is not reusable across different applications. The result shows that the performance of our prototype implementation is comparable to the performance of the hand-tuned implementation. Overall, our mixin-layer based implementation has an average overhead of 1.5% across all the benchmark applications as compared to the hand-tuned optimized implementation, which is a reasonable overhead considering the ease-of-reuse and adaptation advantages provided by our approach.

Next, we study how our implementation performs across our studied configurations. Table 3.8 shows the execution time of our benchmark applications under different resource configurations. We have used the input size of 512 MB for our benchmark applications for each of the resource configurations. The result of executing different applications of various computational densities shows that our implementation scales well with increasing the same kind of accelerator-based compute nodes, as well as using them in a highly heterogeneous environment. Note that between *Conf 1* and *II* (and *Conf III* and *IV*), only the number of accelerator-based compute nodes are increased. In both these scenarios, we increase the number of compute nodes by a factor of $(10/4 =)2.5$, and observe the average speedup by a factor of 2.2 in our prototype implementation. We also observe an increase in the manager workload in distributing the given input and merging the received results from the compute nodes, when we increase the number of compute nodes in *Conf II* and *IV* as compared to

Conf I and *III*, respectively. Similarly, we observe a linear increase in execution time for the benchmark applications in *Conf V* as compared to *Conf I* and *III*.

3.4 Chapter Summary

In this chapter, we have presented CellMR, an extended MapReduce-based programming model, to program accelerator-based heterogeneous clusters. We have discussed the target accelerators and have presented various resource configurations for heterogeneous clusters in conventional and hierarchical settings. We apply CellMR on these resource configurations, and show that it effectively captures the resource configuration complexity while expecting minimal resource management directives from the application programmer. In addition, CellMR provides a clear distinction between the operations of the manager and the compute nodes, and implements these operations in the runtime framework. We evaluate and compare the CellMR and its runtime with traditional, hand-tuned, approaches under different symmetric and asymmetric heterogeneous resource configurations and find it to be more efficient than traditional approaches and highly scalable with increasing number of compute nodes. We have also explored how layered architecture based advanced software engineering approaches can be used to reduce the application deployment time for accelerator-based heterogeneous clusters. We have developed mixin-layers based reusable software components for Cell and GPU-based heterogeneous clusters that effectively encapsulates the manager and compute node functionalities into distinct heterogeneous and platform-specific components. Our evaluation reveals that our mixin-layer based implementation of the software components can be reused with minimal modifications across all the studied heterogeneous accelerator-based clusters and benchmark applications. Furthermore, the performance of the component-based implementation is reasonable, as it has little overhead compared to the hand-tuned and optimized implementations, which are nontrivial to achieve and maintain.

Chapter 4

Improving I/O Performance on Asymmetric Clusters

Asymmetric multi-core processors are widely regarded as a viable path to sustaining high performance, without compromising reliability. This chapter explores the use of the Cell/BE, arguably a dominant asymmetric multi-core processor, in data-intensive applications. With modern high-performance computing applications generating and processing exponentially increasing amounts of data, the scalable parallel processing capabilities, large on-chip data transfer bandwidth, and aggressive latency overlap mechanisms of the Cell/BE render it an attractive platform for high-performance I/O. Although several recent efforts have demonstrated the potential of the Cell/BE for high-speed computation on data staged through its accelerator cores (SPE's) [18,19], there is little understanding of how I/O operations interact with the architecture of the Cell/BE. The implications of such Cell/BE characteristics as asymmetry, DMA latency overlap, and software management of disjoint address spaces, on the design and implementation of the I/O software stack have not been explored.

In this chapter, we describe the techniques that we have explored for improving the I/O performance on asymmetric multicore, specifically the Cell processor. While designing the I/O improvement techniques, our focus is on developing a data staging mechanism for Cell-based asymmetric cluster. We first describe the I/O path that is followed from PPE and SPE while servicing an I/O request in the Cell processor, and elaborate on the I/O characteristics of the Cell processor. We then present and evaluate several schemes to improve I/O performance on the Cell processor.

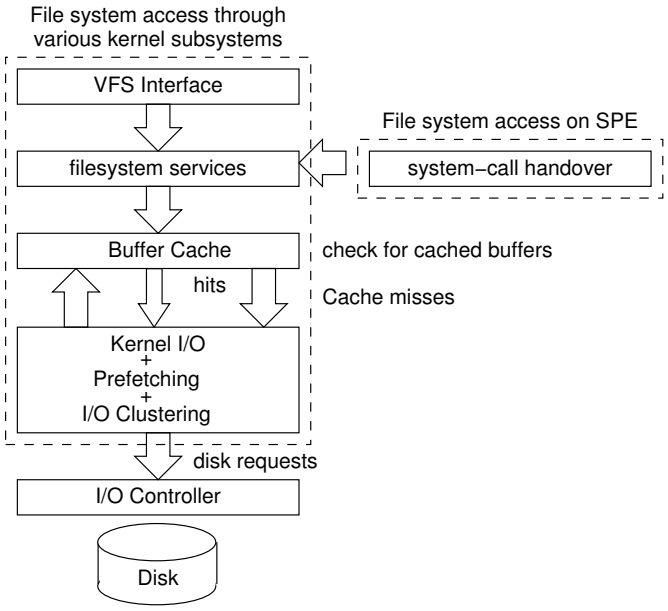


Figure 4.1 I/O requests path on the PPE and SPE.

4.1 PPE/SPE I/O Path in the Cell Broadband Engine

Figure 4.1 shows the path that application I/O requests from the PPE and SPE's follow before being serviced by the disk. The PPE supports a full operating system, and the I/O path on the PPE is a standard one. All I/O requests through the VFS layer are first sent to the buffer cache. In case of miss in the buffer cache, a request to read the data from the disk is issued. The kernel clustering mechanisms combine multiple requests for contiguous blocks, and the kernel prefetching algorithm detects and prefetches blocks to reduce execution stalls due to synchronous disk requests. Interested readers are referred to [218] for a detailed explanation of the standard Linux I/O path.

Since the SPE does not support a native operating system, there is no kernel context on SPE and all system calls issued by SPE's are handled as *external assisted library calls*. We now discuss how external calls are supported. Note that the same process is used to service all system calls from SPE's on the PPE, not just the I/O related calls.

The SPE uses special stop-and-signal instructions [219] to hand over control to the PPE for handling external service requests. In order to perform an external assisted library call, the SPE allocates local store memory to hold the input and output parameters of the call and

copies all the input parameters (from stack or registers) into this memory. It then combines a special function opcode corresponding to the requested service with the address of this input/output memory image to form a 32-bit message. The SPE then places this message into the local store memory, immediately followed by a `stop` instruction. It then signals the PPE to execute the library function on behalf of SPE. In response to the signal, the PPE reads the assisted call message from SPE's local store, and uses the stop and signal type and opcode to dispatch the control (PPE context) to the specified assisted call handler. The handler on the PPE retrieves the input parameters for the assisted call from the local-store memory pointed to by the assisted call message, and executes the appropriate system call on the PPE. On completion of the system call, the return values are placed into the same local store memory, and the SPE is signaled to resume execution. Upon resumption, the library on the SPE reads the input value from the memory image and places them into the return registers, hence, completing the call. Thus all I/O calls on the SPE's are routed through the PPE operating system.

4.2 I/O Characteristics of Cell Broadband Engine

In the following, we first evaluate the I/O characteristics of Cell architecture by running simple workloads, then we explore how the SPE's can be used to handle I/O intensive tasks such as data encryption. Finally, to account for experimental errors the presented numbers represent averages over three different runs unless otherwise stated.

4.2.1 Workload Overview

The workload that we have chosen is a 256-bit encryption/decryption application. Our choice is dictated by the computation intensive component of the encryption and decryption along with the need to do large I/O transfers for reading the input and writing the output. The workload reads a file from the disk, encrypts or decrypts it, and then writes back the results. Given that the PS3 has only about 200 MB of main memory available to user programs, we chose to encrypt a 64 MB file. This allows us to keep the entire file in memory if so needed and isolate the effects of buffer caching etc. We also vectorized the computation phase of our workload to achieve high performance on SPE's, which improved the time taken in the computation phase by 42.1%.

	PPE Time	SPE Time
SPE Context Creation	-	1.46
Program Loading on SPE	-	0.16
Thread Creation on SPE	-	0.104
Buffer Allocation	0.012	0.015
File Reading	48688	48414
Buffer Deallocation	0.012	0.016
Total SPE execution time	-	48806
Total time	49664	49241

Table 4.1 Average time (in msec.) required by major tasks while reading a 2 GB file from the disk on the PPE and a SPE.

Block Size	4 KB		16 KB	
	PPE	SPE	PPE	SPE
Time (msec.)	48674	53779	48688	49414
Throughput (MB/s)	41.09	37.19	41.08	40.48

Table 4.2 The average time and observed throughput for reading a 2 GB file from disk on the PPE and a SPE using different block sizes.

4.2.2 Identity Tests

In the first set of experiments, we created a workload that reads a large file of size 2 GB. We refer to this experiment as the Identity Test. The goal of this Test is to determine the maximum I/O bandwidth available on our experimental platform for the PPE and SPE's.

Table 4.1 shows the timing break down for the Identity Test both on the PPE and a SPE using a block size of 16 KB. Note that context creation, loading, and thread creation are only needed when running the Test on the SPE. It is observed that the time to read the file on the SPE is similar to that on the PPE. The table also shows that the cost of the context loading steps on the SPE is relatively insignificant. However, this cost can become crucial if SPE workloads are repeatedly loaded or if the execution time of the SPE program is small.

Next, we modified the block size to 4 KB, and determined the overall time it would take to perform the Identity Test both on the PPE and the SPE. Table 4.2 shows the results, and comparison with the previous case. We observe that while changing block sizes does not have a significant effect on the PPE I/O throughput, the large block size gives better throughput

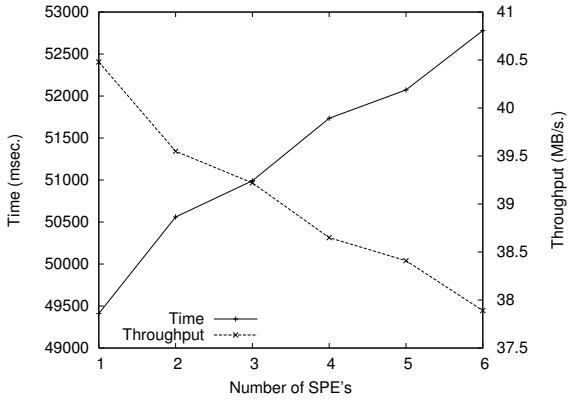


Figure 4.2 Average time and observed throughput for simultaneously reading a 2 GB file using 16 KB blocks from one to six SPE's.

on the SPE. The reason for this improved throughput is that data transfers between SPE and the memory is done via DMA, and DMA is optimized by using the maximum transfer size per DMA operation, which for the Cell/BE is 16 KB. For this reason, in our remaining experiments we set the buffer size to 16 KB.

Next, we repeated the Identity Test while increasing the number of SPE's from one to six. Figure 4.2 shows the result. For this experiment, the PPE invokes one thread on each of the available SPE's, however, the total size of data read is same as before, i.e., 2 GB. A different file is read at each SPE so that unique requests are issued and any caching at the I/O controller and/or memory does not come into play. As seen in the figure, the average observed throughput decreases as the number of SPE's reading the file increases. The average observed I/O throughput is reduced by 6.4% when all the six available SPE's are used, compared to the case of using a single SPE for the same amount of data. This is due to increased contention for the EIB, and indicates that simply offloading I/O intensive jobs to multiple SPE's is unlikely to yield the best use of resources.

4.2.3 Effect of DMA Request Size

The execution model on Cell requires that the computation be offloaded to specific SPE's. Therefore, we first evaluate the effect of DMA buffer sizes on such offloading. Table 4.3 shows the time of computation offloading as we varied the buffer size used for DMA communications between the PPE and SPE. Note that these buffers are different from the file I/O block size of the previous experiments (which is fixed at 16 KB). We focused on the

Num. SPE's	Buffer size				
	1 KB	2 KB	4 KB	8 KB	16 KB
1	21750	13952	9427	7828	6394
6	95410	56953	37031	26168	21206

Table 4.3 Time measured (in msec.) at PPE for sending data to SPE through DMA under varying buffer sizes, and for using one and six SPE's.

decryption phase of our workload for this experiment. In this case, all I/O is performed at the PPE, which after reading a full buffer of data from disk, passes its address in the main memory to a SPE. The SPE uses the passed address to do a DMA transfer and brings the contents of the buffer to its local store. The SPE then processes the data in the local store, and upon completion of the computation issues another DMA to transfer the processed contents back to the main memory. Finally, the PPE can write the updated buffer in the main memory back to the disk. Note that the maximum size of a single channel DMA that can be sent on the EIB is 16 KB, thus the maximum DMA size of our experiments is limited to that. The whole experiment is repeated for two cases: using a single SPE, and using all six SPE's. These results show that increasing the buffer size improves the execution times of our workload.

4.2.4 Timing Breakdown for 4 KB and 16 KB DMA Buffers

For the previously described experiment, we also performed a detailed timing analysis for 4 KB and 16 KB DMA's using a single SPE. Table 4.4 shows the results. This experiment was conducted to see the effect of different DMA sizes on the time spent on various parts of the program. For the same input file, when the DMA size is increased from 4 KB to 16 KB, the number of times the PPE has to invoke a thread on an SPE is reduced by a factor of 4, thus reducing SPE loading time. The number of times the SPE is loaded to perform the same task also affects the total execution time, since it cuts down the number of times initialization is required on the SPE. Table 4.4 shows that the total execution time for the same workload is less when SPE and the PPE communicate with each other through DMA operations and a block size of 16 KB, than using a block size of 4 KB for the same data set. Observe that the total execution time is significantly less when using 16 KB blocks compared to 4 KB blocks. This is due to the fact that the total time also includes the time required at SPE to fetch the data into its local store through DMA operations, and the number of

	Time (msec.)	
Buffer size	**4 KB**	**16 KB**
Number of times SPE is loaded	16384	4096
SPE loading (excluding execution) time	1787	823
SPE execution (including loading) time	8014	4273
CPU time used by SPE	5200	1850
Disk read time	450	497
Disk write time	1191	1221
CPU time for disk read operations	400	570
CPU time for disk write operations	330	250
Execution time of program	10176	6565
CPU time used by PPE	6050	2890

Table 4.4 Breakdown of time spent (in msec.) in different portions of the code when data is exchanged between a SPE and the PPE through DMA buffer sizes of 4 KB and 16 KB.

	PPE	**PPE**	**SPE**	**SPE**
Time	3403	205	714	640
	SPE	**SPE**	**PPE**	**PPE**
Time	4174	329	217	217

Table 4.5 Time (in msec.) for reading workload file at PPE/SPE followed by access from SPE/PPE.

DMA operations done by SPE for 16 KB blocks is 4 times less than that for 4 KB blocks for the same data set.

4.2.5 Impact of File Caching

As discussed in Section 4.1, the I/O system calls from the SPE are handed over to the PPE for handling. This implies that once a file (or portion of a file) is accessed by the PPE it may be in memory when subsequent access for the file are issued from a SPE or the PPE, and these accesses can be serviced fast. In this experiment, we aim at confirming this empirical observation. First, we flushed any file cache by reading a large file (2 GB). Then we read the 64 MB workload file on the PPE, followed by reading the same file at a SPE. Table 4.5 shows the result for reading a file cold first on the PPE, followed by reading at SPE. The same experiment is repeated for first reading the file at a SPE, followed by at the PPE. From the table, we conclude that the caching effect is noticeable, and can help in reducing I/O

times both on the PPE and on the SPE's, by first reading a file on the PPE. We also notice that file reading on the SPE is slower due to the I/O being routed through the PPE.

4.3 Memory-Layout and I/O Optimization Techniques for Cell Architecture

Given the effectiveness of file caching, we have explored a number of schemes to improve I/O performance of our workload. Figure 4.3 shows the results. In some schemes tasks are executed in parallel at the PPE and SPE's. This is shown as two side-by-side bars for a scheme, with the total execution time dictated by the higher of the two bars. The breakdown for various steps is also shown. In the following we describe these schemes in detail.

4.3.1 Scheme 1: SPE Performs All Tasks

Under this scheme, we perform all the tasks of our workload, i.e., reading the input file (b), processing it (d), and writing the output file (f), on the SPE. Note, however, that we still utilize the PPE to invoke the tasks as a single program on the SPE.

4.3.2 Scheme 2: Synchronous File Prefetching by the PPE

In this scheme, we attempt to improve the overall performance of our workload by allowing the PPE to prefetch the input file in memory. This scheme is driven by the above observation that subsequent accesses by SPE's to a file read earlier by the PPE improves I/O times due to file caching. For this purpose, the PPE first pre-reads the entire file causing it to be brought in memory. Then the program from Scheme 1 is executed as before. Results in Figure 4.3 show that the *File read at SPE* (b) is much faster for this scheme, compared to Scheme 1. However, the time it takes to read the file on the PPE (a) is 81.6% longer compared to *File read at SPE* (b) in Scheme 1. We believe this is due to the PPE flooding the I/O controller queue, and lack of overlapping opportunities between computation and I/O in a sequential read compared to the read and process cycle of Scheme 1. Hence, Scheme 2 shows promise in terms of improving SPE read times, but suffers from slow I/O times on the PPE. The overall workload execution time is longer in Scheme 2 than Scheme 1.

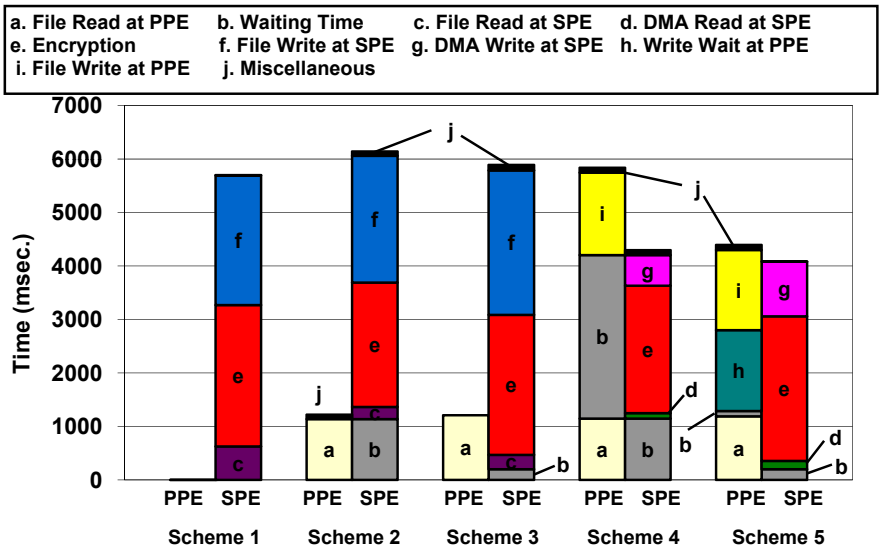


Figure 4.3 Timing breakdown of different tasks for five data transfer schemes.

4.3.3 Scheme 3: Asynchronous Prefetching by the PPE

In the next scheme, we try to remove the file reading bottleneck of Scheme 2. For this purpose, we created a separate thread to prefetch the file into memory. Simultaneously, we offloaded the program of Scheme 1 to the SPE. The goal is to allow the prefetching by the PPE to overlap with computation on SPE, thus any data accessed by SPE will already be in memory and the overall performance of the workload will improve. Note that we do not have to worry about synchronizing the prefetching thread on the PPE with the I/O on SPE. In case the PPE thread is ahead of SPE, no problems would arise. However, if the SPE gets ahead of the PPE thread, the SPE's I/O request will automatically cause the data to be brought into memory, which in turn will make the PPE read the file faster, thus once again getting ahead of the SPE. The integrity of data read by SPE will not be compromised.

It is observed from the results in Figure 4.3 that although the I/O times (a) for individual steps increased, better I/O/computation overlapping resulted in an overall improvement of 4.7%, compared to Scheme 2. This shows that the PPE can facilitate I/O for SPE's and doing so results in improved performance.

4.3.4 Scheme 4: Synchronous DMA by the SPE

The schemes presented so far attempts to improve SPE performance by indirectly bringing the file in memory and implicitly improving the performance of the SPE workload. However, such schemes are prone to problems if the system flushes the file read by the PPE from the buffer cache before it can be read by SPE, hence negating any advantage of a PPE-assisted prefetch.

In this scheme, we explicitly prefetch the file on the PPE and give the SPE the address of memory where the file data is available. The SPE program is modified to not do direct I/O, rather use the addresses provided by the PPE. Hence, the PPE will read the input file in memory, give its address to the SPE to process, the SPE will create the output in memory, and finally the PPE will write the file back to the disk. The SPE will use DMA to map portions of the mapped file to its local store and send the results back. Figure 4.3 shows the results. Here, we observe that the *DMA read at SPE* (c) takes 55.0% and 62.0% less time than *File read at SPE* (b) in Scheme 2 and Scheme 3, respectively. However, the synchronous reading of file in this scheme takes long, causing the overall times to not improve as much: 4.9% and 0.2% compared to Scheme 2 and Scheme 3, respectively.

4.3.5 Scheme 5: Asynchronous DMA by the SPE with Signaling

The main shortcoming in Scheme 5 is the lack of a signaling mechanism between the prefetching thread producing the data (reading into memory) and the SPE consuming the data. One way to address this to use the *mailbox* abstraction supported by the Cell/BE. However, documentation [220] advises against using *mailboxes* given their slow performance. Therefore, we used DMA-based shared memory as a signaling mechanism to keep the prefetching thread synchronized with the SPE's. The PPE starts a thread to read the input file, and simultaneously also starts the SPE process. The difference from Scheme 5 is that the prefetching thread continuously updates a *status location* in main memory with the offset of the file read so far, and uses this location to determine how much of the data has been produced by SPE for writing back to the output file. Moreover, the SPE process, instead of blindly accessing memory assuming it contains valid input data, periodically uses DMA to access a pre-specified memory *status location*. In case the prefetching thread is lagging, the SPE process will busy-wait and recheck the *status location* until the required data is loaded into memory. Finally, the SPE can also use the shared location to specify the amount of pro-

cessed output. This allows the PPE to simultaneously write back the output to the disk, and achieve an additional improvement over Scheme 5 where output was written back only after the entire input was processed. Thus, Scheme 6 achieves both reading of the input file and writing of output file in parallel with the processing of the data. Figure 4.3 shows the results, which are quite promising. Scheme 6 achieves 22.2%, 24.1%, and 24.0% improvement in overall performance compared to Scheme 1, Scheme 3, and Scheme 4, respectively.

4.4 Chapter Summary

In this chapter, we have investigated prefetching-based techniques for supporting data intensive workloads involving significant computation components on the Cell architecture. We study the data path to and from the general-purpose (PPE) and specialized (SPE) cores within the Cell architecture. We have presented and evaluated different prefetching techniques for the Cell processor, and have shown that the asynchronous prefetching techniques, where the PPE prefetches the data into the main memory for SPE, can effectively eliminate the I/O bottlenecks from the Cell processor.

Chapter 5

Capability-Aware Workload Distribution for Heterogeneous Clusters

While the potential of many-core accelerators to catalyze HPC is clear, attempting to integrate heterogeneous resources seamlessly in large-scale computing installations raises challenges, with respect to managing heterogeneous resources and matchmaking computations with resource characteristics. The trend towards integrating relatively simple cores with extremely efficient vector units, leads to designs that are inherently compute-efficient but control-inefficient. As such, the capabilities of many-core accelerators to run control-intensive code, such as an operating system, or a communication library, are inherently limited. To address this problem, large-scale system installations use ad hoc approaches to pair accelerators with more control-efficient processors, such as x86 multicore CPUs [21], whereas processor architecture moves in the direction of integrating control-efficient and compute-efficient cores on the same chip [6]. Using architecture-specific solutions is highly undesirable in both cases, because it compromises productivity, portability, and sustainability of the involved systems and applications.

In Chapter 3, we have presented a solution that addresses the challenges of programmability for asymmetric accelerator-based clusters. In this chapter, we extend CellMR to address the challenges of memory limitations in using heterogeneous resources. We introduce enhancements in three aspects of the MapReduce programming model presented in Chapter 3: (a) We exploit accelerators with techniques that improve data locality and achieve overlapping of MapReduce execution stages; (b) We introduce runtime support for exploiting multiple accelerator architectures (Cell and GPUs) in the same cluster setup and adapting workload task execution to different accelerator architectures at runtime; (c) We introduce

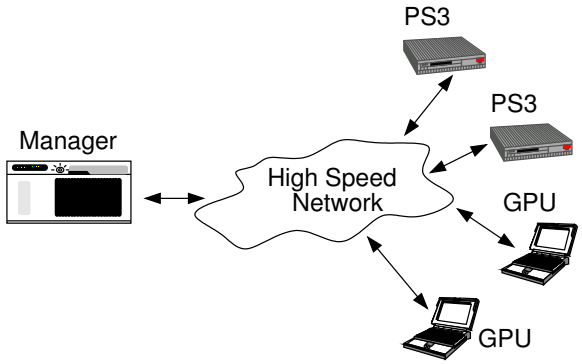


Figure 5.1 High-level overview of a Cell and GPU-based heterogeneous cluster.

workload-aware execution capabilities for virtualized application execution setups. The latter extension is important in computational clouds comprising heterogeneous computational resources, where effective and transparent allocation of resources to tasks is essential.

5.1 Heterogeneous System Architecture

Figure 5.1 shows the heterogeneous cluster architecture that we explore in this chapter. A general purpose multicore server acts as a dedicated front-end *manager* for the cluster and manages a number of back-end accelerator-based nodes. The manager is responsible for scheduling jobs, distributing data, allocating work between compute nodes, and providing other support services at the front-end of the cluster. Accelerator nodes provide high-performance data processing capabilities to the cluster. To isolate our exploration from the impact of the numerous optimizations available on each accelerator-type processor, we assume that readily optimized, architecture-specific executable code for the different components of the application is available for all types of accelerators, for example, through vendor-optimized libraries. In our experimental setup, this code would typically be available through accelerator-specific programming toolkits, such as CUDA [4], and Cell SDK [220].

The manager divides the MapReduce components (map, reduce, partitioning and sorting) into small tasks suitable for parallel execution. It then invokes the associated binaries on the accelerators and assigns the tasks to accelerators for data processing and aggregation. If the back-end is a Cell-based compute node, its generic core uses MapReduce within the node, to map the assigned workload to the accelerator cores (SPEs). If the back-end is GPU-based, its generic x86 core uses MapReduce to execute the assigned workload to the attached GPU. When compute nodes complete execution of their respective workloads, the manager collates

the results, performs any application-specific data merging needed, and produces the final result. The manager has the option to offload part of the data merging workload operations to accelerators as necessary.

Our framework uses transparently optimized accelerator-specific binaries on the accelerators. In this way, the runtime hides the asymmetry between different available resources. Nevertheless, a given application component will exhibit variation in performance on the different combinations of processor types, memory systems, and node interconnects available on the cluster. To improve resource utilization and matchmaking between MapReduce components and available hardware resources, the runtime system monitors the execution time of tasks on hardware components and uses this information to adapt the scheduling of tasks to components, so that each task ends up executing on the resource that is best suited for it. The application programmer may also guide the runtime by providing an *affinity* metric that indicates the best resource for a given task, e.g., a high affinity value for a GPU implies that an application component would perform best on a GPU, whereas an affinity of zero implies that the application should preferably execute on other types of processors. The runtime system takes these values into consideration when making its scheduling decisions.

5.2 Efficient Application Data Allocation

Efficient allocation of application data to compute nodes is a central component in our design. This poses several alternatives. A straw man approach is to simply divide the total input data into as many chunks as the number of available processing nodes, and copy the chunks to the local disks of the compute nodes. The application on the compute nodes can then get the data from the local disk as needed, and write the results back to the local disk. When the task completes, the result-data can be read from the disk and returned to the manager. This approach is easy to implement, and lightweight for the manager node as it reduces the allocation task to a single data distribution.

Static decomposition and distribution of data among local disks can potentially be employed for well-provisioned compute nodes. However, for nodes with small memory, there are several drawbacks: (i) it requires creation of additional copies of the input data from the manager's storage to the local disk, and vice versa for the result data, which can quickly become a bottleneck, especially if the compute node disks are slower than those available to the manager; (ii) it requires compute nodes to read required data from disks, which have greater

latency as compared to other alternatives, such as main memory; (iii) it entails modifying the workload to account for explicit copying, which is undesirable as it burdens the application programmer with system-level details, thus making the application non-portable across different setups; (iv) it entails extra communication between the manager and the compute nodes, which can slow the nodes and affect overall performance. Hence, this is not a suitable choice for use with small-memory accelerators.

A second alternative is to still divide the input data as before, but instead of copying a chunk to the compute node's disk as in the previous case, map the chunk directly into the virtual memory of the compute node. The goal here is to leverage the high-speed disks available to the manager and avoid unnecessary data copying. However, for small-memory nodes, this approach can create chunks that are very large compared to the physical memory available at the nodes, thus leading to memory thrashing and reduced performance. This is exacerbated by the fact that available MapReduce runtime implementations [31] require additional memory reserved for the runtime system to store internal data structures. Hence, static division of input data is not a viable approach for our target environment.

The third alternative is to divide the input data into chunks, with sizes based on the memory capacity of the compute nodes. Chunks should still be mapped to the virtual memory to avoid unnecessary copying, whereas the chunk sizes should be set so that at any point in time, a compute node can process one chunk while streaming in the next chunk to be processed and streaming out the previously computed chunk. This approach can improve performance on compute nodes, at the cost of increasing the manager's load, as well as the load of the compute node cores that run the operating system and I/O protocol stacks. Therefore, we seek a design point which balances the manager's load, I/O and system overhead on compute nodes, and raw computational performance on compute nodes. We adopt this approach in our design.

5.3 Capability-Aware Workload Scheduling

We consider two types of accelerators, Cell processors and CUDA-enabled GPUs and design a scheduler that handles both stand-alone and virtualized execution of applications. In the latter case, applications share resources in space and/or time. The scheduler takes two parameters as input: (i) the number and type of compute nodes in the heterogeneous cluster; and (ii) the number of simultaneously running applications on the heterogeneous cluster. In

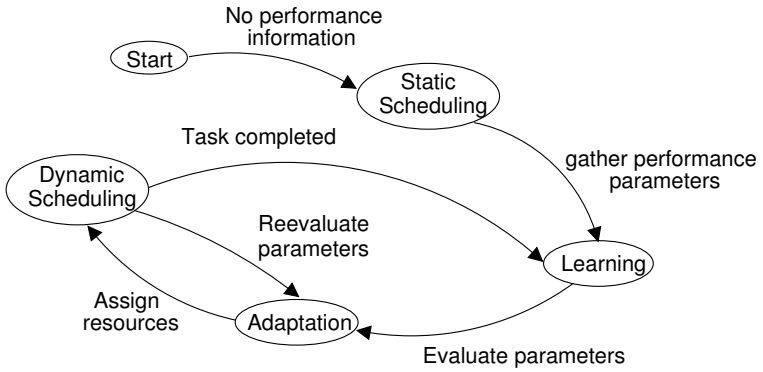


Figure 5.2 State machine for scheduler learning and execution process.

the following, we first describe different execution states of the scheduler, and then present the scheduling algorithm.

5.3.1 Scheduling States

Figure 5.2 shows the different states representing the learning process and the execution flow of the scheduler. Initially, the scheduler starts with a static assignment of tasks to nodes and processors, based on the user-provided affinity metric, the performance of the resources in terms of time spent per byte of data or if no information is available by simply dividing the tasks evenly between resources. The scheduler then enters its learning phase, where it measures the processing times for different application components on the resources on which they are initially scheduled. Based on the processing time of the workload on each of the available compute nodes, the scheduler then computes the processing time per byte for each of the available compute nodes. Once a processing rate is known, the scheduler moves to the adaptation phase, where the schedule is adjusted so as to greedily maximize the processing rate. Note that, even in this phase, the scheduler continues to monitor its performance and adjust its scheduling decisions accordingly.

For simultaneously executing multiple applications, the scheduler must decide which application to run on what particular accelerator. For this purpose, the scheduler tries different assignments, e.g., starting by scheduling an application A on the Cell processor and application B on the GPU for a pre-specified period of time T_{learn}, then reversing the assignment for another T_{learn}, determining the assignment that yields higher throughput, and finally using that assignment for the remaining execution of the application. The time to determine a best schedule will increase with the number of applications executing simultaneously, however, it

Algorithm 5.1: Capability-aware workload scheduling.

```
Input: nodeArray, appArray
for node ∈ nodeArray do
    for app ∈ appArray do
        estArray = sendNextChunk(node, app);
    end
end
nodeAppMap = GenNodeAppComb(estArray, nodeArray, appArray);
while incompleteAppExist(nodeAppMap) do
    for appNode ∈ nodeAppMap do
        app = getApp(appNode);
        node = getNode(appNode);
        if appCompleted(app) then
            nodeAppMap = GenNodeAppComb(estArray, nodeArray, appArray);
        end
        else
            estArray = sendNextChunk(node, app);
        end
    end
end
```

can be reduced by using user input or information from past application runs. If one of the applications completes earlier than the other, the scheduler enters the learning phase and attempts to assign the newly freed resources to either applications waiting in the queue, or to the running application if the queue is empty. We note that it is not always possible to assign the applications to the most suitable nodes, e.g., when multiple applications need the same type of accelerators and only a limited number of the accelerator is available. Nonetheless, our approach ensures that all the compute nodes are kept busy, and that the assignment of the applications to the compute nodes is optimal in that the overall completion time for the scheduled applications is minimized for the given resources and applications.

5.3.2 Scheduling Algorithm

Algorithm 5.1 describes how our capabilities-aware dynamic scheduling scheme works. Since the scheduler does not have any information about how the available compute nodes perform for the given tasks, a static schedule is chosen in the beginning. The schedule is adjusted dynamically as the tasks execute and their performance on the assigned compute nodes can

be measured. Note that if there are more accelerators available than the number of applications, each application is scheduled on a separate accelerator during the learning phase. This eliminates any resource conflicts between different applications and allows for determining accurate processing rates. For example, if applications A and B are assigned to a cluster with four each of Cell- and GPU-based compute nodes, the system assigns half of the nodes (2 Cells and 2 GPUs) each to A and B during the learning phase. The assignment is then readjusted using the performance measurements to improve the overall execution time.

5.4 Addressing Manager-Accelerator I/O Mismatch

Even with adaptive scheduling and programmer-supplied affinity metrics, the inherent asymmetry between cluster components may lead to performance degradation, especially due to communication delays associated with data distribution and collection by the manager. Thus, it is critical to handle all communication with different components of the system asynchronously. This design choice needs careful consideration. If chunks from consecutive input data are distributed to multiple compute nodes, it would require time-consuming complex sorting and ordering to ensure proper merging of the results from individual compute nodes into a consolidated result set. We address this issue by using a separate handler thread on the manager for each of the compute nodes. Each handler works with a consecutive fixed portion of the data and avoids costly ordering operations by exploiting data locality. Each handler thread is also responsible for receiving all the results from its associated compute node, and for performing the application-specific merging operation on the received data. This design leverages multicore or multi-processor head nodes effectively. Moreover, we use well-established techniques such as prefetching and double buffering to avoid I/O delays when reading data from the disks, and transferring the data between manager and compute nodes.

5.5 Dynamic Work Unit Scaling

Efficient utilization of compute nodes is crucial for overall system performance. A key observation in CellMR is that a compute node's performance can be increased many fold by reducing memory pressure, which is in turn tied to the work unit size. A very large work unit results in thrashing on the compute nodes, while an unnecessarily small work unit increases the workload of the manager. In either case, system performance is reduced. The challenge

is finding an optimally-sized work unit, which offers the best trade-off between the compute node performance and manager load.

An optimum work unit for running an application on a particular cluster can be manually determined by hard coding different work unit sizes, executing the application, and measuring the execution time for each size. The best work unit size is the one for which the application execution time is minimized. However, this is a tedious and error-prone process, and requires unnecessary "test" access to resources, which is difficult to obtain given the ever increasing need for executing "production" tasks on a cluster to maintain high serviceability.

To remedy this, CellMR provides the manager with the option to automatically determine the best work unit size for a particular application. This is done by sending compute nodes varying work unit sizes at the start of the application and recording the completion time corresponding to each work unit. A binary search technique is used to modify the work unit size to determine one that gives the highest processing rate calculated using $(work\ unit\ size)/(execution\ time)$. If the processing rate is the same for two work unit sizes, the larger one is preferred as that minimizes load on the manager. The determined work unit size is chosen as the most efficient for use with the application and employed for the rest of the application run.

All available compute nodes participate in finding the optimal work unit size. The manager sends increasing work units to multiple compute nodes simultaneously, although one size is sent to at least two compute nodes to determine average performance. Once the optimal work unit size is determined, it can also be reported to the application user to possibly facilitate optimization for a future run.

5.6 Evaluation

In this section, we present our experimentation platforms, the benchmarks that we use to evaluate our framework in heterogeneous settings, and the results of our experiments along with their analysis.

5.6.1 Experimental Setup

We build an experimental testbed using 4 Sony PlayStation 3 (PS3), 4 GPU enabled Toshiba Qosmio laptops, and a manager node. All cluster components are connected via 1 Gbps

Ethernet in our testbed.

Manager Node The manager has two quad-core Intel Xeon processors with 3 GHz clocks, 16 GB main memory, and 650 GB hard disk. The manager node runs Linux Fedora Core 8 (kernel version 2.6.26).

Cell-based PS3 Node The PS3 has eight Synergistic Processing Elements, out of which six are available to user-level programs [45, 201], 256 MB of main memory (of which about 200 MB is available for applications and the rest is reserved for the operating system and a proprietary hypervisor), and a 60 GB hard disk. The PS3 node further has a swap space of 512 MB and runs Linux Fedora Core 7 (kernel version 2.6.24).

GPU-based Node The GPU enabled Toshiba Qosmio laptops have one Intel dual-core processor with a 2.0 GHz clock, 4 GB of DRAM, and run Linux Fedora Core 9 (kernel version 2.6.27). Each of the laptops also has one GeForce 9600M GT CUDA enabled GPU device [217], with 32 cores and 512 MB of memory. We program the GPU using the CUDA toolkit 2.2.

5.6.2 Raw Performance Comparison of Cell and GPU Nodes

Table 5.1 shows the raw performance comparison of Cell and GPU nodes used in our evaluation. It can be seen that the peak performance of each component of GPU nodes is considerably lower than that of PS3. Overall, considering all computational units on PS3 and GPU nodes, the GPU-enabled laptops are 1.31× slower than the Cell-based PS3 nodes in terms of peak performance.

5.6.3 Methodology

We focus our evaluation on the design decisions for accelerator-based clusters and on deriving clues about the best way to utilize a given set of accelerator-based resources for maximizing overall performance. We also evaluate the performance of our capabilities-aware scheduling scheme for simultaneously running multiple applications on heterogeneous resources. To this end, we compare the performance of our dynamic scheduling scheme with a static scheduling scheme. The static scheme takes into account the overall performance of the assigned workloads on Cell and GPU nodes, which in essence incorporates all the performance pa-

Cell Node		
PowerPC Clock Speed	3.2 GHz.	
SPE Clock Speed	3.2 GHz.	
PowerPC Peak Performance	25.6 GFLOPS.	
SPEs Peak Performance	153.6 GFLOPS.	
On-chip Mem. Bandwidth	204.8 GBps.	
Mem. Interface Controller	25.6 GBps.	
GPU Node		**Wrt. Cell**
Intel Clock Speed	2.0 GHz.	62.5%
GPU Clock Speed	1.25 GHz.	39.1%
Intel Peak Performance	16.0 GFLOPS.	62.5%
GPU Peak Performance	120.0 GFLOPS.	78.1%
GPU Internal Mem. Bandwidth	25.6 GBps.	12.5%
PCI-Express Bandwidth	8.0 GBps.	31.2%

Table 5.1 Performance comparison of Cell and GPU nodes.

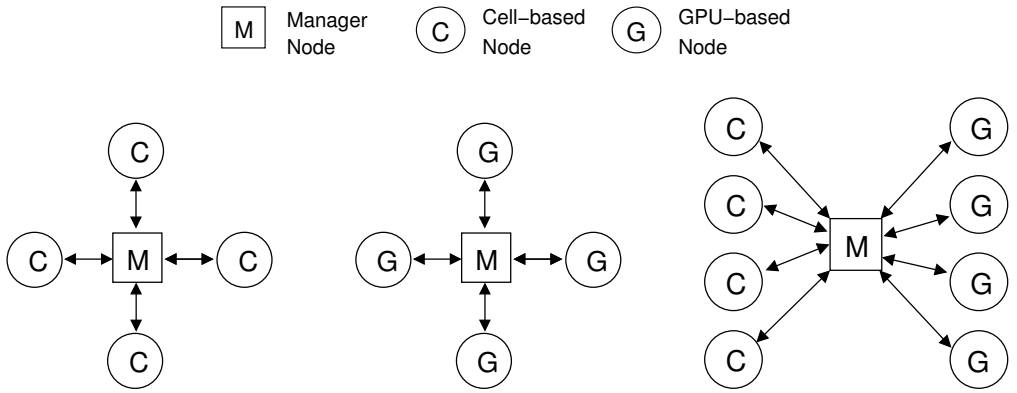
rameters shown in Table 5.1, thus providing the best static scheduling scheme for the given applications on the studied platforms.

5.6.3.1 Applications

We have used well-known parallel applications namely Linear Regression, Word Count, Histogram, and K-Means described in Section 3.2.4 to evaluate the performance of our capability-aware workload scheduling in a heterogeneous cluster. These applications are commonly used in scientific computing environments, including epidemiology, environmental science, image segmentation, and statistical analysis [202–204] and have characteristics that are representative of many parallel applications.

5.6.3.2 Resource Configurations

Figure 5.3 shows three different resource configurations that we have used in our evaluation. *PS3 Cluster* (Figure 5.3(a)) and *GPU Cluster* (Figure 5.3(b)) configurations have 4 PS3 nodes and 4 GPU nodes connected to the manager node, respectively. The *PS3+GPU Cluster* (Figure 5.3(c)) configuration has all the 8 accelerator-nodes (4 PS3 and 4 GPU nodes) connected with the manager node.



(a) *PS3 Cluster:* Cell-based cluster.

(b) *GPU Cluster:* GPU-based cluster.

(c) *PS3+GPU Cluster:* Cell and GPU-based cluster.

Figure 5.3 Resource configurations for Cell and GPU based heterogeneous clusters.

5.6.4 Results

We first examine how the studied benchmarks behave on alternative cluster configurations. Table 5.2 shows the execution time for running the four benchmarks on a stand-alone PS3 and GPU node without using our framework. It is observed that for the PS3 node, Linear Regression is the only benchmark that successfully completes for all of the specified input sizes. For the GPU node, all benchmarks incur swapping and run out of swap space and available memory for input sizes greater than 192 MB. Furthermore, the observed performance difference of executing these benchmarks on the PS3 node versus the GPU node is higher than the corresponding raw performance. This is because CUDA-enabled GPUs use stream processing with each CUDA thread executing the same instructions but on different data. If some threads take a different execution path on a branch, the execution of such threads is serialized. The map and reduce phases of Word Count, Histogram and K-Means have many branch instructions, which cause some threads to stall while others to continue their execution on different execution paths. Thus, reducing overall performance of these benchmarks on the GPUs. On the other hand, each SPE of the Cell processor in the PS3-node can process the assigned data chunk independent of other SPEs without stalling. This gives an additional performance advantage to the PS3 node over the GPU node. However, for both accelerator types, the execution time for all the benchmarks increases rapidly with the increasing input size without utilizing our framework. Thus, these compute nodes cannot be simply plugged into traditional clusters to achieve high performance. An integrated management framework with capabilities-aware scheduling across heterogeneous compute nodes

Input (MB)	Linear Regression		Word Count		Histogram		K-Means	
	PS3	GPU	PS3	GPU	PS3	GPU	PS3	GPU
4	0.34	0.35	1.95	4.28	1.06	22.15	1.66	5.46
64	2.88	40.65	-	1396.56	45.66	390.67	167.93	509.54
128	12.56	81.65	-	-	318.66	1805.98	-	-
192	21.81	231.98	-	-	394.78	-	-	-
256	34.89	-	-	-	-	-	-	-

Table 5.2 Execution time (in seconds) on a single PS3 and GPU node.

is necessary to address this problem.

5.6.4.1 Benchmark Performance

Linear Regression For this benchmark, the input size ranges from 2^{21} points (file size 2 MB) to 2^{29} points (512 MB). Figure 5.4 shows the results. All the resource configurations show similar scaling patterns with the increasing input size. Overall *PS3 Cluster* performs 11.4% better than *GPU Cluster*. Note that although the PS3 node is 1.31× faster than the GPU in terms of GFLOPS, the speedup achieved in terms of overall execution time improvement is much less. The lower gains are due to the fact that Linear Regression is a map-dominated application and spends a significant portion of its time on map operations. These operations require DMA transfers from DRAM to local accelerator memory. The GPU has 512 MB of device memory and the needed 4 MB data chunk is transferred in a single DMA operation. In contrast, PS3's SPE local stores are only 256 KB (for holding the executable, data, etc.) and the most efficient DMA transfer size is 16 KB [45], consequently it requires a large number of DMA operations. Although we attempt to overlap DMA transfers with the computation as explained in Section 5.4, these DMA operations consume cycles for setting up data transfers, therefore even though the PS3 node has much better raw performance than the GPU, most of the PS3 node's advantage is negated due to it having small per-core local store. Overall, our framework effectively utilizes the combined heterogeneous resources in *PS3+GPU Cluster*. On average, the 8 computational nodes in *PS3+GPU Cluster* perform 46.9% and 53.0% better than the 4 computational nodes in *PS3 Cluster* and *GPU Cluster*, respectively.

Table 5.3 shows the percentage of time spent in different MapReduce stages by PS3 and GPU nodes when a chunk is assigned to them. Note that both types of compute nodes show similar trends in terms of the percentage of time spent at each MapReduce stage.

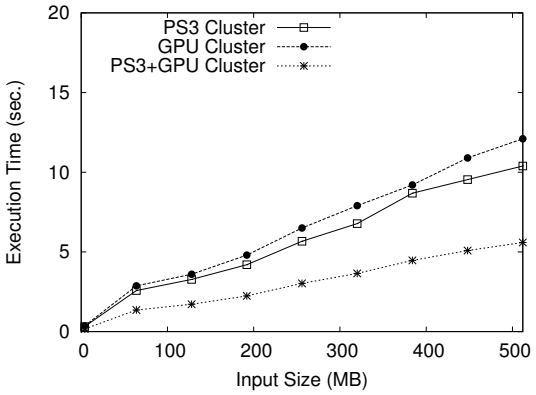


Figure 5.4 Execution time of Linear Regression with increasing input size on heterogeneous cluster.

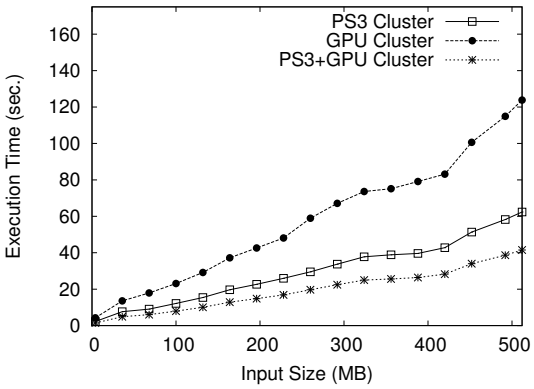


Figure 5.5 Execution time of Word Count with increasing input size on heterogeneous cluster.

Application Name	Partition		Map		Sort		Reduce	
	PS3	GPU	PS3	GPU	PS3	GPU	PS3	GPU
Linear Regression	1.3	2.1	95.1	93.3	2.4	1.0	1.2	3.6
Word Count	87.1	86.1	3.0	7.5	9.4	4.3	0.5	2.1
Histogram	75.2	56.1	3.5	3.0	20.8	40.1	0.5	0.8
K-Means	40.3	42.1	55.2	52.4	2.4	2.2	2.1	3.3

Table 5.3 Distribution of time (in %) spent in different stages of MapReduce for both the PS3 and GPU based clusters.

Word Count The input partitioning in Word Count is a serial operation performed on the main core of the compute nodes and is not parallelized between the SPE/GPU cores. Here, we observe an exponential growth in memory consumption relative to the input data size in both PS3 and GPU nodes, since each word would emit additional intermediate data out of the map function. This has a direct impact on execution time as shown in Table 5.2. For any input size greater than 44 MB, a single accelerator node thrashes and runs out of available swap space (512 MB).

Figure 5.5 shows the result of running Word Count using our framework under the studied resource configurations. We vary the input size from 2 MB to 512 MB. All resource configurations show linear increase in execution time with increasing input size. For this benchmark, on average, *PS3+GPU Cluster* performs 34.1% and 65.9% better than *PS3 Cluster* and *GPU Cluster*, respectively. Further analysis reveals that, on average, *PS3 Cluster* performs 48.1% better than *GPU Cluster*.

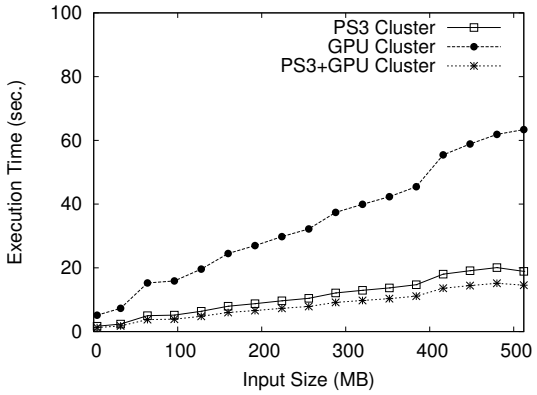


Figure 5.6 Execution time of Histogram with increasing input size on heterogeneous cluster.

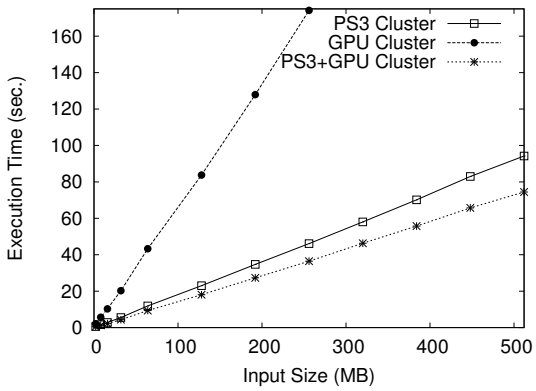
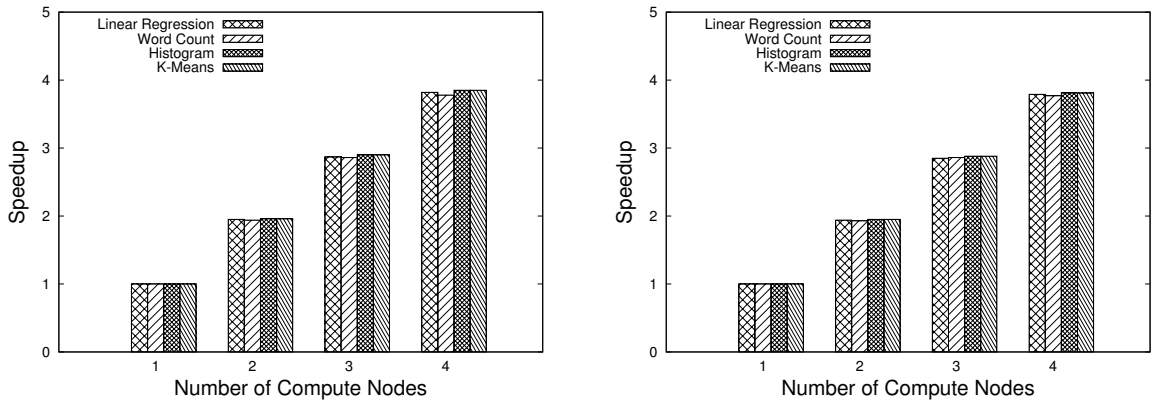


Figure 5.7 Execution time of K-Means with increasing input size on heterogeneous cluster.

In Word Count, the amount of data offloaded to the SPE and GPU is 4 KB and 2 MB, respectively. The size of the data offloaded to the SPE and GPU is dynamically computed at runtime so as to give optimal overall processing time (including map, reduce and merge time) on each compute node. It also enables each SPE and GPU to process the data without overwhelming their local stores or device memory, respectively. As shown in Table 5.3 both the Cell and GPU implementations of Word Count spend more than 86% of their time in partitioning the input and intermediate data between the accelerator cores. PS3 nodes outperform GPU nodes because of the higher sustained GFLOP rates and faster intra-node interconnects.

Histogram Figure 5.6 shows results from executions of Histogram under the three test configurations. On average *PS3+GPU Cluster* performs 24.4% and 75.6% better than *PS3 Cluster* and *GPU Cluster*, respectively. Similarly, on average *PS3 Cluster* performs 67.7% better than *GPU Cluster*, since Histogram requires excessive sorting operations (as shown in Table 5.3) on the intermediate data that incurs additional penalty on GPUs, e.g., synchronization with the other CUDA threads at each iteration. On PS3 nodes parallel sort is performed on the SPE, which is controlled by the PPE, resulting in less execution stalls in each iteration. In this benchmark, the amount of data offloaded to the SPE and GPU is 2 KB and 0.6 MB, respectively, which is computed dynamically at runtime to optimize the data processing performance of each accelerator.



(a) PS3 Cluster. (b) GPU cluster.

Figure 5.8 Speedup with increasing compute nodes in PS3 and GPU based cluster for studied benchmarks.

K-Means Figure 5.7 shows the results of experiments with K-Means. K-Means uses a different number of iterations for different input sizes. Therefore, considering total execution times for different inputs does not provide a fair comparison of the effect of increasing input size. We remedy this by reporting the execution time per iteration in the figure. The result for this benchmark shows that *PS3+GPU Cluster* performs 21.2% and 78.8% better than *PS3 Cluster* and *GPU Cluster*, respectively. Moreover, *PS3 Cluster* performs 73.0% better than the *GPU Cluster*. In K-Means, the amount of data offloaded to the SPE and GPU is 3 KB and 1 MB, respectively.

5.6.4.2 Speedup with Increasing Number of Compute Nodes

Next, we observed how the studied benchmarks scale with the increasing number of nodes in PS3-based and GPU-based clusters. Figure 5.8 shows the achieved speedup of executing the benchmarks normalized to the corresponding 1-node cluster for PS3-based and GPU-based cluster setting. In this experiment, the input size is set to 512 MB for the studied benchmarks under PS3-based and GPU-based cluster configurations. The result of this experiment shows that our implementation scales almost linearly for both the PS3-based and GPU-based clusters for all the benchmark applications.

5.6.4.3 Scheduling Multiple Applications on Available Resources

In the next set of experiments, we invoked multiple applications on the manager node to simulate a cloud computing environment where multiple applications are assigned to the cluster and computational resources are shared transparently between applications. The goal here is to see how well our scheduler assigns the job to the compute nodes based on the performance of each type of compute node for the given application.

We compare our dynamic scheduling with a static scheduling scheme that simultaneously schedules all applications to be run on all the compute nodes. The static scheduling scheme uses knowledge about how the applications would perform on each type of the compute nodes and how much data can be handled by the nodes at a time, i.e., the amount of memory the nodes have available, to divide and assign to the nodes the input data to be processed. In contrast, our dynamic scheduler has no prior knowledge of the nodes' capabilities, and learns and adapts as the applications proceed.

Word Count and Histogram In this experiment, we simultaneously executed Word Count and Histogram jobs, with input data of 512 MB each, on *PS3+GPU Cluster*, with 4 PS3 and 4 GPU nodes, and observe how these two benchmarks are scheduled on PS3 and GPU nodes based on the capabilities of individual compute nodes. As shown in the earlier experiments, PS3 nodes execute Word Count 48.1% faster than GPU nodes. Similarly, Histogram is executed 67.7% faster on PS3 node than on GPU nodes.

Figure 5.9 shows the result of this experiment with static and dynamic scheduling of compute nodes to the given tasks. In static scheduling, both the benchmarks are executed on the PS3 and GPU nodes. However, most of the execution is carried out by the PS3 nodes because it has a higher GFLOP performance, higher on-chip memory bandwidth, and faster memory-to-chip interfaces compared to the GPU nodes as indicated in Table 5.1. Overall, about 68.7% and 97.6% of Word Count and Histogram data, respectively, is processed by PS3 nodes and the remaining by the GPU nodes.

In contrast, the dynamic scheduler is quickly able to learn that assigning PS3 nodes to Histogram and the GPU nodes to Word Count is more beneficial. Specifically, Word Count on GPU node and Histogram on PS3 node take 53.9s and 20.3s to complete, respectively. Conversely, Word Count on PS3 node and Histogram on GPU node take 61.2s and 62.3s, respectively. The former assignment is 13.6% and 206.9% better for Word Count and His-

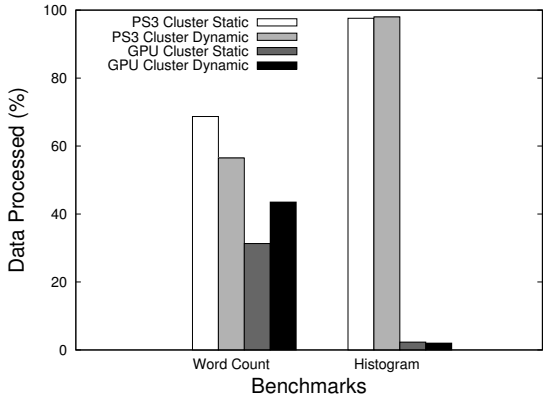
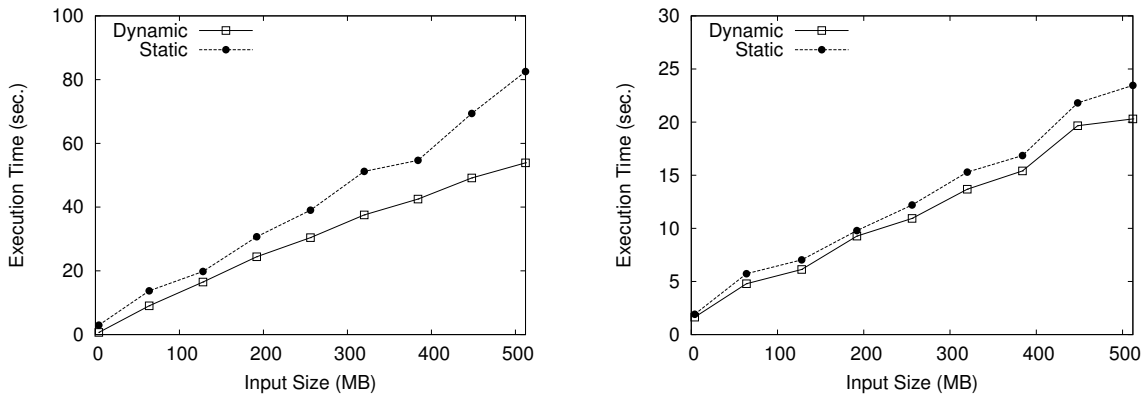


Figure 5.9 Percentage of data processed on PS3 and GPU nodes for simultaneously running Word Count and Histogram using static and dynamic scheduling schemes.

togram performance, respectively, and is thus chosen by our scheduler.

Once the execution of Histogram is completed, the scheduler includes the PS3 nodes in available resources for the Word Count application, and assigns the remaining data to both PS3 and GPU node. The result of this experiment shows that 98.0% and 56.5% of Histogram and Word Count data, respectively, is processed by PS3 nodes. Conversely, 2.0% and 43.5% of Histogram and Word Count data, respectively, is processed by the GPU nodes. Compared to static scheduling, 12.2% more of Word Count data is processed at the GPU nodes under the dynamic scheduling scheme. Note that the Histogram completes soon after the learning phase that also uses static scheduling. This accounts for why only a small (0.4%) increase in the amount of Histogram data processed at the PS3 node is observed between static and dynamic scheduling.

Figure 5.10 shows the execution time for simultaneously running Word Count and Histogram using the static and dynamic scheduling with increasing input sizes. For static scheduling, both the benchmarks are executed on all the resources, and completion of an application does not affect the allocation of resources for other applications. For dynamic scheduling, although the benchmarks start to execute together, Histogram completes quickly, leaving Word Count to utilize all the available resources for its remaining execution. Overall, compared to static scheduling, our dynamic scheduling scheme performs 31.5% and 11.3% better for Word Count and Histogram, respectively.



(a) Execution time for Word Count.

(b) Execution time for Histogram.

Figure 5.10 Execution time for simultaneously running Word Count and Histogram.

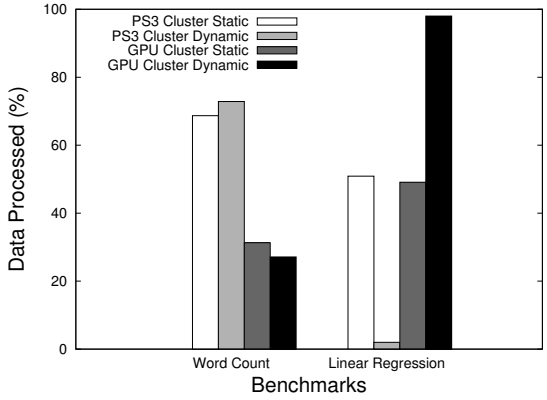
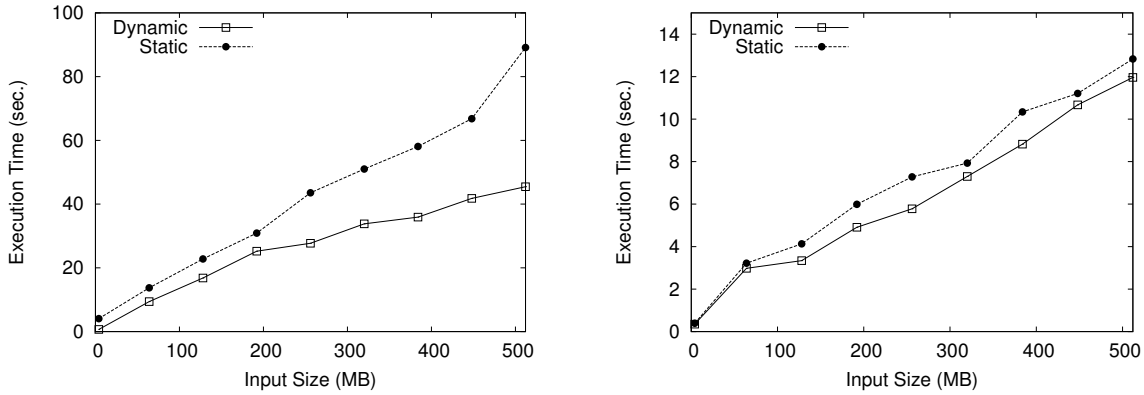


Figure 5.11 Percentage of data processed on PS3 and GPU nodes for simultaneously running Word Count and Linear Regression using static and dynamic scheduling schemes.

Word Count and Linear Regression In this experiment, we simultaneously executed Word Count and Linear Regression jobs, with input data of 512 MB each, on *PS3+GPU Cluster*, with 4 PS3 and 4 GPU nodes. As shown in earlier experiments, PS3 nodes execute Word Count 48.1% faster than GPU nodes. Similarly, Linear Regression is executed 11.4% faster on PS3 node than on GPU nodes.

Figure 5.11 shows the result of simultaneously running Word Count and Linear Regression with static and dynamic scheduling of compute nodes.

In case of static scheduling, both the benchmarks are executed on the PS3 and GPU nodes. Both types of compute nodes execute both benchmarks during their entire execution lifecy-



(a) Execution time for Word Count.

(b) Execution time for Linear Regression.

Figure 5.12 Execution time for simultaneously running Word Count and Linear Regression.

cle, and process the data for each benchmark based on their respective capabilities. In this case, 68.7% and 50.9% of Word Count and Linear Regression data, respectively, is processed by PS3 nodes. Similarly, 31.3% and 49.1% of Word Count and Linear Regression data, respectively, is processed by the GPU nodes.

In contrast, the dynamic scheduling scheme schedules resources based on the components capabilities to execute a particular benchmark. Since the performance advantage of running Word Count on PS3 nodes is more than executing Linear Regression on PS3 nodes, the scheduler of our framework schedules Word Count on PS3 nodes, while executing Linear Regression on the GPU nodes. Once the execution of Linear Regression is completed on the GPU nodes, the scheduler divides the remaining unprocessed data for Word Count between the PS3 and GPU nodes and assigns the remaining data to both PS3 and GPU nodes based on their processing capabilities. The result of this experiment shows that 72.8% and 2.0% of Word Count and Linear Regression data, respectively, is processed by PS3 nodes. Conversely, 27.1% and 98.0% of Word Count and Linear Regression data, respectively, is processed by the GPU nodes.

Figure 5.12 shows the execution time for simultaneously running Word Count and Linear Regression with increasing input sizes using the static and dynamic scheduling schemes. Overall, our dynamic scheduling outperforms the static scheduling schemes by 39.3% and 12.5% for Word Count and Linear Regression, respectively.

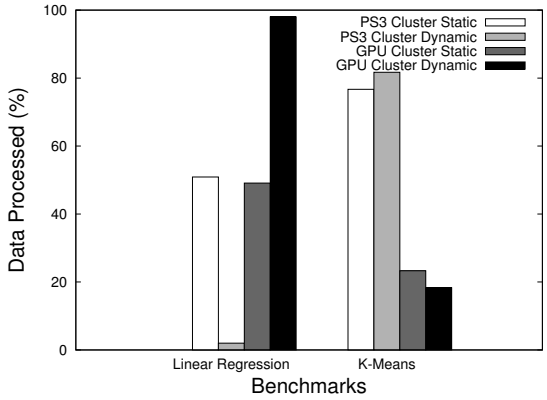


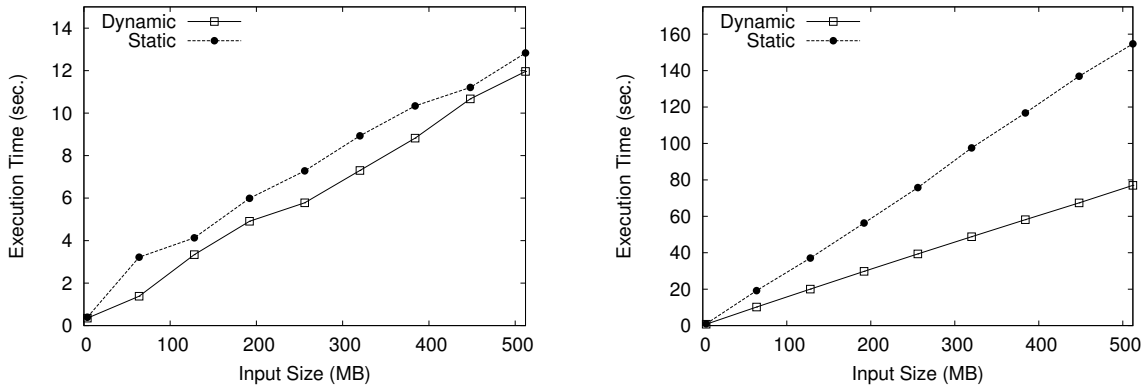
Figure 5.13 Percentage of data processed on PS3 and GPU nodes for simultaneously running Linear Regression and K-Means using static and dynamic scheduling schemes.

Linear Regression and K-Means In this experiment, we simultaneously executed two map-intensive jobs, i.e. Linear Regression and K-Means, with input data of 512 MB each, on *PS3+GPU Cluster*, with 4 PS3 and 4 GPU nodes, and observed how our framework schedules these benchmarks on the available compute nodes. Note that we have shown earlier that Linear Regression and K-Means execute 11.4% and 73.0% faster on a PS3 node compared to a GPU node, respectively.

Figure 5.13 shows the results for both static and dynamic scheduling. In case of static scheduling, both types of compute nodes execute both the benchmarks during the entire execution lifecycle of the benchmarks. In this case, 50.9% and 76.7% of Linear Regression and K-means data, respectively, is processed by PS3 nodes. Similarly, 49.1% and 23.3% of Linear Regression and K-Means data, respectively, is processed by the GPU nodes.

In the case of dynamic scheduling, our scheduler exploits the capabilities of individual nodes for executing the benchmarks. Here, Linear Regression is scheduled on the GPU-cluster because it provides better performance on GPUs than K-Means. K-Means is executed on the PS3 cluster until Linear Regression is execution, however, once Linear Regression completes, the scheduler utilizes all resources, i.e., PS3- and GPU-cluster, for K-Means to expedite the overall execution. Overall, PS3 nodes process 2.0% and 81.7% of Linear Regression and K-Means data, respectively, while GPU nodes process 98.0% and 18.3% of Linear Regression and K-Means data, respectively.

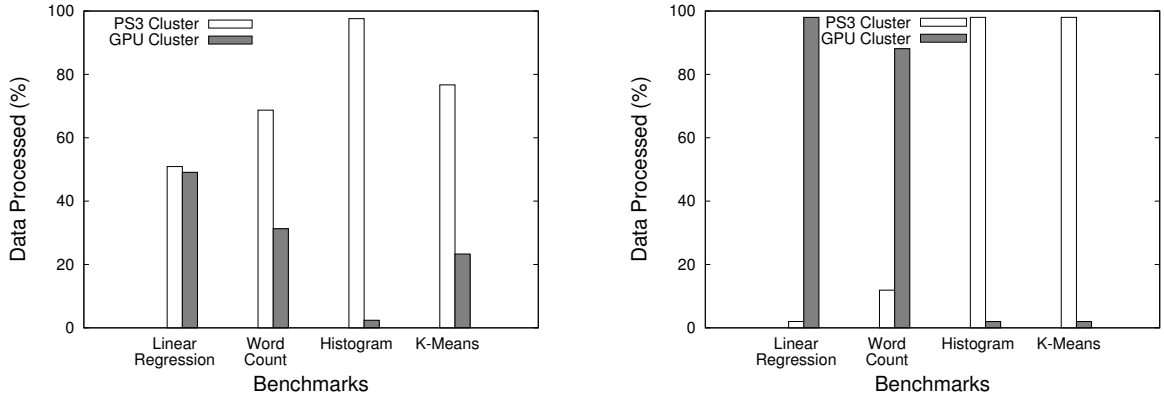
Figure 5.14 shows the execution time for simultaneously running Linear Regression and K-



(a) Execution time for Linear Regression.

(b) Execution time for K-Means.

Figure 5.14 Execution time for simultaneously running Linear Regression and K-Means.



(a) Static scheduling of benchmarks to the compute nodes.

(b) Dynamic scheduling of benchmarks to the compute nodes.

Figure 5.15 Percentage of data processed on PS3 and GPU nodes for simultaneously running the studied benchmarks.

Means using the static and dynamic scheduling schemes. Overall, our dynamic scheduling outperforms the static scheduling schemes by 19.1% and 45.7% for Linear Regression and K-Means, respectively.

All Benchmarks In this experiment, we simultaneously executed all of our benchmarks with the input data of 512 MB each, on the *PS3+GPU Cluster* and observed the performance of our scheduler compared to the static scheduling for these benchmarks. Figure 5.15 shows the results. In case of static scheduling, shown in Figure 5.15(a), all of the benchmarks are scheduled simultaneously on all compute nodes. In this case, PS3 nodes process

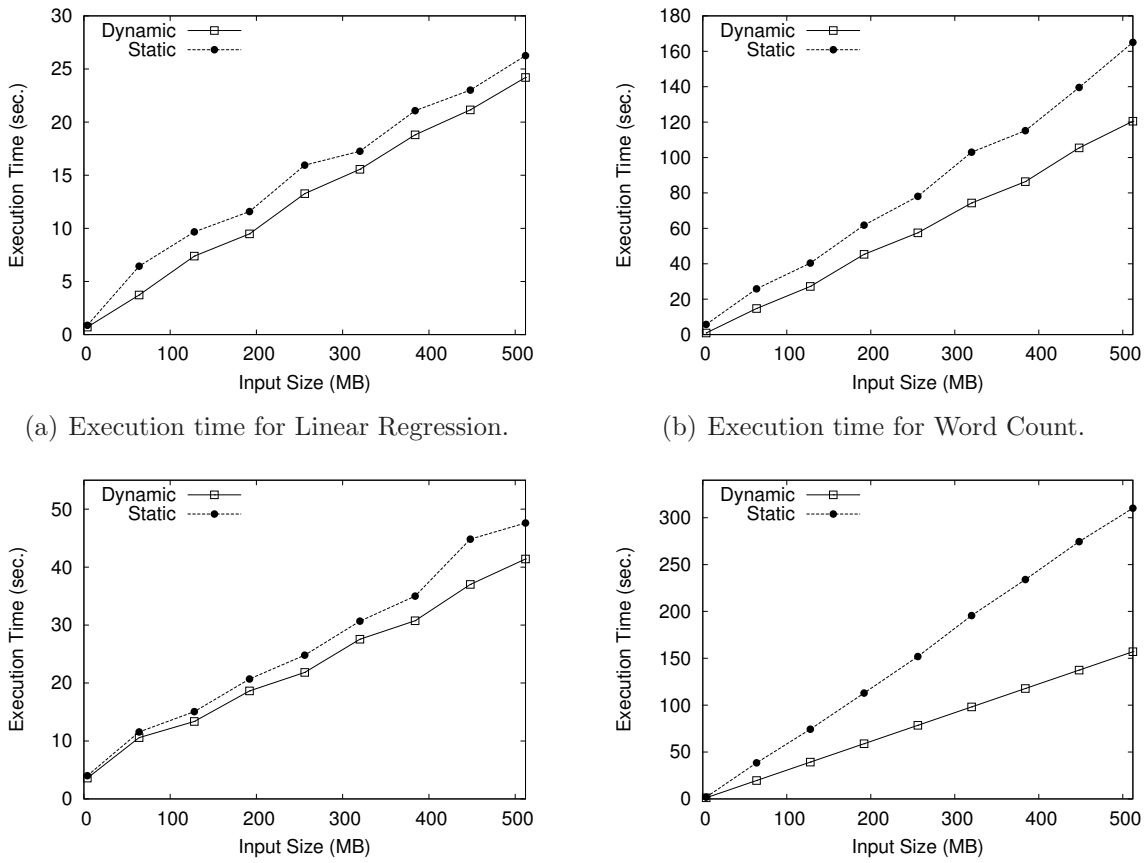
50.9%, 68.7%, 97.6% and 76.7% of Linear Regression, Word Count, Histogram, and K-Means respectively. Similarly, the GPU nodes process 49.1%, 31.3%, 2.4% and 23.3% of Linear Regression, Word Count, Histogram, and K-Means data respectively. In the case of dynamic assignment of resources to applications, shown in Figure 5.15(b), our scheduler takes advantage of the relative capabilities of the cluster nodes for each benchmark: it schedules Linear Regression and Word Count on the GPU nodes, while Histogram and K-Means are scheduled on the PS3 nodes. This way, Linear Regression completes earlier than the other benchmarks, enabling our scheduler to start scheduling the unprocessed Word Count data between the PS3 and GPU nodes. The next benchmark which completes its execution is Histogram, which leaves K-Means executing on the PS3 nodes and Word Count on the PS3 as well as GPU nodes. Overall, in the case of dynamic scheduling, the PS3 nodes process 2.0%, 11.9%, 98.0%, and 98.0% of Linear Regression, Word Count, Histogram, and K-Means data, respectively. Conversely, the GPU nodes process 98.0%, 88.1%, 2.0%, and 2.0% of Linear Regression, Word Count, Histogram, and K-Means data, respectively.

Figure 5.16 shows the execution time for simultaneously running all the studied benchmarks with increasing input sizes using the static and dynamic scheduling schemes. Overall, our dynamic scheduling outperforms the static scheduling schemes by 17.3%, 35.2%, 11.7% and 47.4% better for Linear Regression, Word Count, Histogram and K-Means respectively.

5.6.5 Work Unit Size Determination

As discussed earlier in Section 5.5, the work unit size affects the performance of compute nodes, and consequently the whole system. In this experiment, we first show how varying work unit sizes affect the processing time on a node. For this purpose, we use a single PS3 node connected to the manager, and run Linear Regression with an input size of 512 MB[1]. Figure 5.17 shows that as the work unit size is increased, the execution time first decreases to a minimum, and eventually increases exponentially. The valley point (shown by the dashed line) indicates the size after which the compute node starts to page. Using a larger size reduces performance. Using a size smaller than this point wastes resources: notice that the curve is almost flat before the valley indicating no extra overhead for processing more data. Also, using a smaller work unit size increases the manager's load, as the manager now has to handle larger number of chunks for a given input size. Using the valley point work unit size is optimal as it provides the best trade-off between compute node's and manager's

[1]The results are the similar in other applications and input sizes.



(a) Execution time for Linear Regression.

(b) Execution time for Word Count.

(c) Execution time for Histogram.

(d) Execution time for K-Means.

Figure 5.16 Execution time for simultaneously running all the benchmarks.

performance, and results in minimal execution time.

Next, we evaluate CellMR's ability to dynamically determine the optimal work unit size. In principle, the optimal unit size depends on the relative computation to data transfer ratios of the application and machine parameters, most notably, latencies and bandwidths of the chip, node and network interconnects. We follow an experimental process to discover optimal work unit size. We manually determined the maximum work unit size for each application that can run on a single PS3 without paging to be the optimal work unit size. We compared the manual work unit size to that determined by CellMR at runtime. For each application, Table 5.4 shows: the work unit size both determined manually and automatically, the number of iterations done by CellMR to determine the work unit, and the time it takes for the reaching this decision. Our framework is able to dynamically determine an appropriate work unit that is close to the one found manually, and the determination on

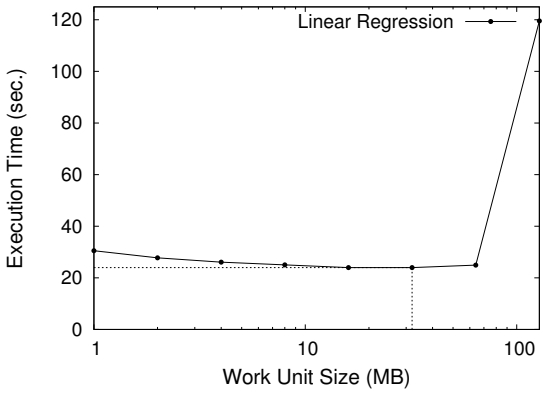


Figure 5.17 Effect of work unit size on execution time.

Application	Hand-Tuned Size (MB)	CellMR		
		Size (MB)	# Iterations	Time (s)
Linear Regression	32	30	16	0.65
Word Count	3	2	8	1.82
Histogram	2	1	4	0.15
K-Means	0.37	0.12	16	1.09

Table 5.4 Performance of work unit size determination.

average across our benchmarks takes under 0.93 seconds. This is negligible, i.e., less than 0.5% of the total application execution times when input size is 2 GB. Note that optimal work unit size determination is independent of the given input size, and has a constant cost for a given application. Thus, dynamic work unit scaling in our framework is efficient as well as reasonably accurate.

5.6.6 Impact on the Manager

In this experiment, we determine the affect of varying work unit sizes on manager performance. This is done as follows. First, we start a long running job (Linear Regression) on the cluster. Next, we determine the time it takes to compile a large project (Linux kernel 2.6) on the manager, while the MapReduce task is running. We repeat the steps as the work unit size is decreased, potentially increasing the processing requirements from the manager. For each work unit size, we repeat the experiment 10 times using symmetric heterogeneous cluster, and record the minimum, maximum, and average time for the compilation as shown

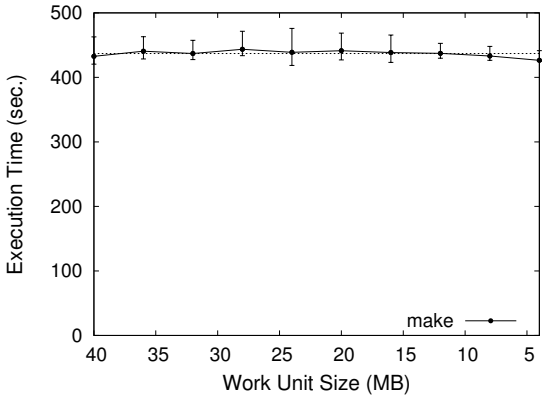


Figure 5.18 Impact of work unit size on the manager.

in Figure 5.18. The horizontal dashed line in the figure shows the overall average of average compile time across all work unit sizes. Given that the overall average remains within the minimum and maximum times, we can infer that the variations in the compile time curve is within the margin of error. Thus, the relatively flat curve indicates that CellMR has a constant load on the manager and our framework can support various workloads without the manager becoming a bottleneck.

5.7 Chapter Summary

In this chapter, we have presented different data distribution alternatives for allocating the application data between heterogeneous resources, and have developed an efficient data distribution approach that divides the application data between asymmetric resources based on the memory capacities of individual accelerators and streams the resulting data chunks to each compute node. We have also addressed the challenges of asymmetric resource capabilities by developing a capability-aware workload distribution technique for concurrently executing multiple applications on heterogeneous resources. The proposed mechanism takes into consideration the I/O characteristics of individual application and the memory resources available to each compute node while executing concurrent applications on the heterogeneous cluster. Our experimental results show that the proposed approaches significantly improve the application performance compared to the static data allocation and workload distribution techniques, and enable our framework to efficiently execute data-intensive applications on heterogeneous many-core clusters.

Chapter 6

Energy-Aware Resource Scheduling for Heterogeneous Clusters

Reducing energy consumption has a significant role in mitigating the total cost of ownership (TCO) of computing clusters. Building heterogeneous clusters by combining high-end and low-end server nodes (e.g., Xeons and Atoms) is a recent trend towards achieving energy-efficient computing. However, this requires a cluster-level resource scheduling that has the ability to predict future load, and server nodes that can quickly transition between active and low-power sleep states. In this chapter, we explore the energy benefits of using heterogeneous server nodes, and provide an energy-aware resource scheduling scheme that identifies the optimal cluster configuration based on the power profiles of the heterogeneous nodes and workload characteristics, and maximizes work done per watt by dynamically assigning low power states to the nodes based on the current request rate.

Dynamic Voltage and Frequency Scaling [221] (DVFS) is a popular power optimization technique. Typically each server node runs a default policy (such as on-demand) that scales the frequency of the processor based on operating system performance counters. The power consumption of the CPU constitutes a portion of the total power consumption of a system; therefore the gains from DVFS (i.e., P-states) are relatively small as compared to low power sleep states (i.e., S-states). However, in order to provision for peak load which is hard to anticipate, and due to high wake-up times, servers are typically kept awake. Idle power for servers is usually more than 50% of the peak power. High idle power together with hard-to-predict load spikes limits the effectiveness of sleep states. The energy-aware resource scheduling mechanism presented in this chapter uses P-states and S-states to reduce the energy consumption of the cluster and maximize the work done per watt by the cluster. We observe that performing DVFS at the cluster level and combining it with sleep states yields better energy efficiency than using the either approach in isolation or at the node level.

Application	CPU Frequency (GHz.)	Peak Power (W)	Rate (req/ sec.)	Response Time (ms)	Efficiency (throughput per W)
MediaWiki	1.0	12.9	7	391.0	0.54
MediaWiki	1.5	14.5	10	263.0	0.69
Dynamic Content Server	1.0	12.6	105	20.6	8.33
Dynamic Content Server	1.5	14.2	180	14.2	11.2

Table 6.1 Power/performance profile of Atom N550 machines for the two workloads, Errors/Violations=0

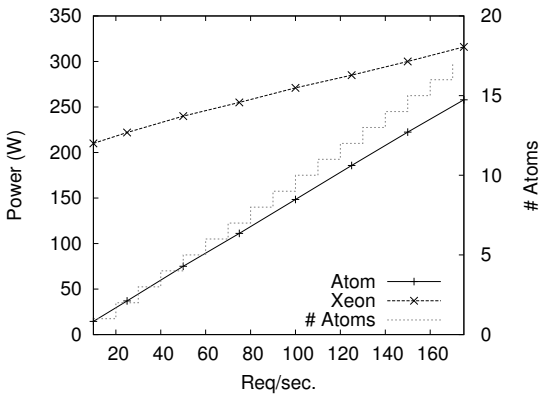
Application	CPU Frequency (GHz.)	Peak Power (W)	Rate (req/ sec.)	Response Time (ms)	Efficiency (throughput per W)
MediaWiki	1.6	257.3	110	186.0	0.43
MediaWiki	2.4	316.0	175	110.0	0.55
Dynamic Content Server	1.6	248.5	2250	10.5	9.05
Dynamic Content Server	2.4	281.1	3000	8.9	10.67

Table 6.2 Power/performance profile of Xeon E5620 machines for the two workloads, Errors/Violations=0.

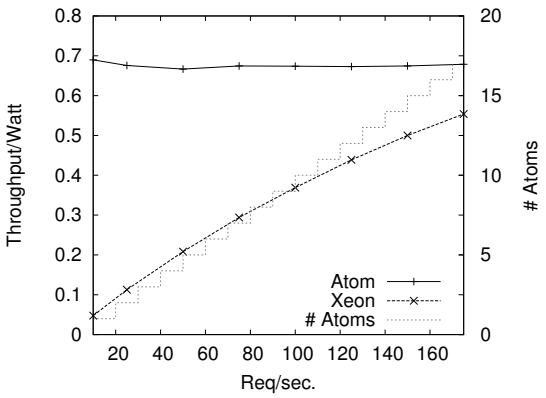
6.1 Motivational Data

In this chapter, we use a heterogeneous cluster consisting of Atom N550-based Netbooks and Xeon E5620-based servers. The Atom N550 [222] is the most recent in the line of Atom processors and the only one to support DVFS. It runs at two frequencies: 1.0 GHz. and 1.5 GHz. Table 6.1 shows the power-performance profile of Atom N550 with respect to the two web workloads, MediaWiki and Dynamic Content Server. Table 6.2 shows the power-performance profile of the Xeon server for the same workloads. The power range of Xeon (difference between peak and idle power) is much higher as compared to Atom for the two workloads. Similarly the throughput range of Xeon is much higher than that of Atom. This motivates a scale-out approach for Atom's (adding Atom servers incrementally to handle increasing load).

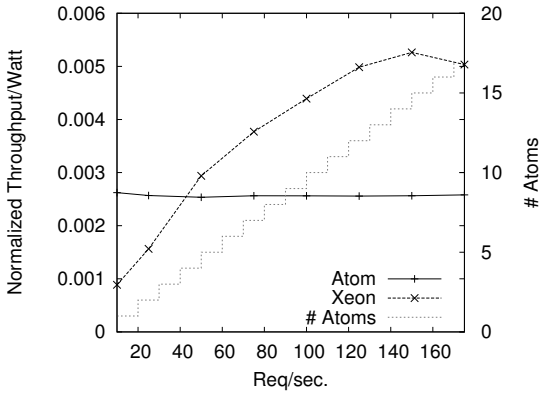
Figure 6.1(a) and 6.2(a) show the raw power consumption of Atom and Xeon as a function of increasing load (requests per second) for the two applications (such that errors/violations=0). As we increase the request rate, a single Atom server cannot sustain the QoS. Therefore, we scale out and provision more Atom servers; the number of Atoms provisioned



(a) Power consumption



(b) Energy efficiency

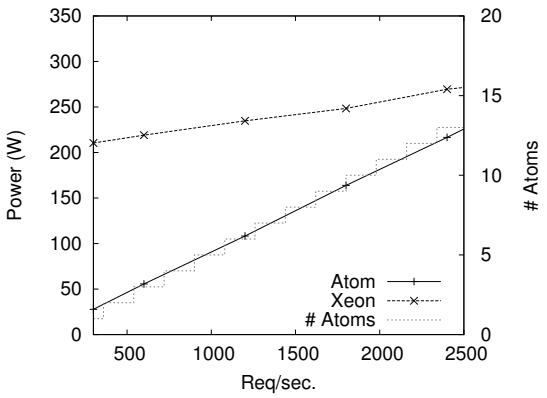


(c) Energy efficiency normalized by response time

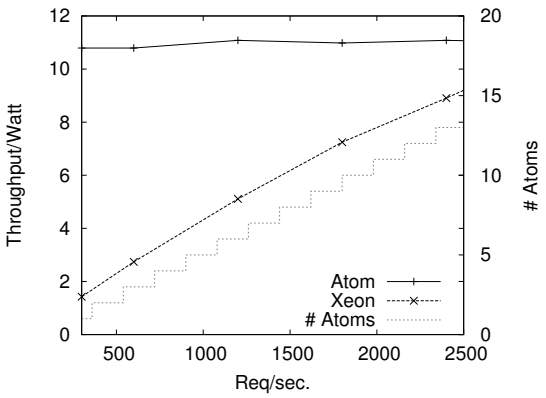
Figure 6.1 Power and energy profiles of Atom and Xeon clusters for MediaWiki.

is shown on the right vertical axis. Even with scaling out, the Atoms come out on top in terms of power consumption. When we consider energy efficiency (throughput per watt) shown in Figure 6.1(b) and Figure 6.2(b), we see that as request rate increases, Xeon starts to catch up with Atom, although the Atom stays ahead.

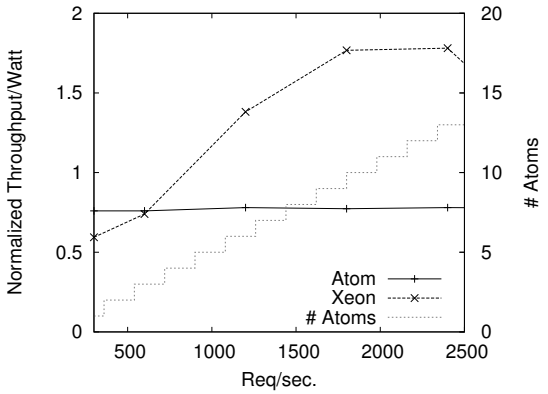
The idle power of Xeon is about 14 times that of the Atom, while the energy efficiency of Atom (throughput per watt) is much higher than that of the Xeon (as shown in Figure 6.1(b) and 6.2(b)). This makes Atom-based servers good candidates for handling light web server workloads. However, as Table 6.1 and Table 6.2 show, Atom has a significantly higher latency (response time per request) than the Xeon for both workloads. Since this has implications on both QoS as well energy consumption, we normalize energy efficiency by response time. This is shown in Figure 6.1(c) and Figure 6.2(c). For low request rates, Atom



(a) Power consumption



(b) Energy efficiency



(c) Energy efficiency normalized by response time

Figure 6.2 Power and energy profiles of Atom and Xeon clusters for Dynamic Content Server.

does better than Xeon on this metric, while the Xeon performs better for high request rates. This motivates heterogeneous clusters for the two workloads (explained in Section 6.3).

There is a significant difference in the maximum throughput being handled by both Atom and Xeon at low and high clock frequency. The efficiency (throughput per watt) is higher at peak frequency for both. This would suggest that consolidating and directing requests to a few servers and running them at higher frequency would result in good energy efficiency. This would certainly be true if the unused servers could be put into hibernation or standby mode. However, in order to provision for unforeseen load spikes, a certain number of servers would have to be kept awake. The total power consumption in this case would be obtained by adding the power consumed by the heavily utilized servers and the power consumed by the

idle servers. The other strategy would be to run the servers at low frequency and distribute the workload among them. We compare different strategies in Section 6.4.

Minimizing the power spent on idle servers by keeping the minimum number of servers awake for handling spikes would improve energy efficiency [223]. Since the idle power of our Xeon is 14 times that of the Atom, it would be profitable to keep Atom-based servers awake and put Xeon-based servers to sleep. When a spike comes, the idle Atom servers can handle the increasing load, while a Wake-on-Lan signal is sent to Xeon servers. Prior studies [224, 225] have shown that it usually takes a few minutes for a load spike to peak. Xeon servers take around 90 sec. to be brought up from standby. Therefore, in order to plan for load spikes, we need enough idle Atom-based servers to handle the load till the Xeons are brought back into operational state.

6.2 System Architecture

Figure 6.3 shows the power manager architecture. *Input Handler* receives client requests and stores them in a request queue. *Policy Manager* is the central component. It interacts with the other components and implements the power management policy for the cluster. It reads the requests from the queue, and redirects them to the appropriate server. During the initialization process, *Policy Manager* invokes *Profiler and Cluster Configurator*, which profiles the given workload on the cluster, generates lookup tables and identifies the optimal cluster configuration (described later in this section). *DVFS Driver* remotely sets the appropriate CPU frequency on a given server node. *Standby/Hibernate Driver* implements the functionality for putting servers to standby mode and waking them over LAN. *Load Analyzer* interacts with the *Input Handler* and monitors the rate of incoming user requests. The *Policy Manager* periodically polls the *Load Analyzer* to detect any spikes in the incoming user requests.

We study two web server workloads that are commonly hosted in a data center. The first workload is a web server application serving dynamic content from the local filesystem. This workload is hosted on a single tier. The second is the MediaWiki+MySQL application. MediaWiki and MySQL are hosted by two separate server tiers; we focus on the front-end tier running the MediaWiki. We consider lightweight workloads to make sure that the QoS/latency is within acceptable bounds, since prior studies [226] has shown that Atom is not suitable for computationally intensive workloads.

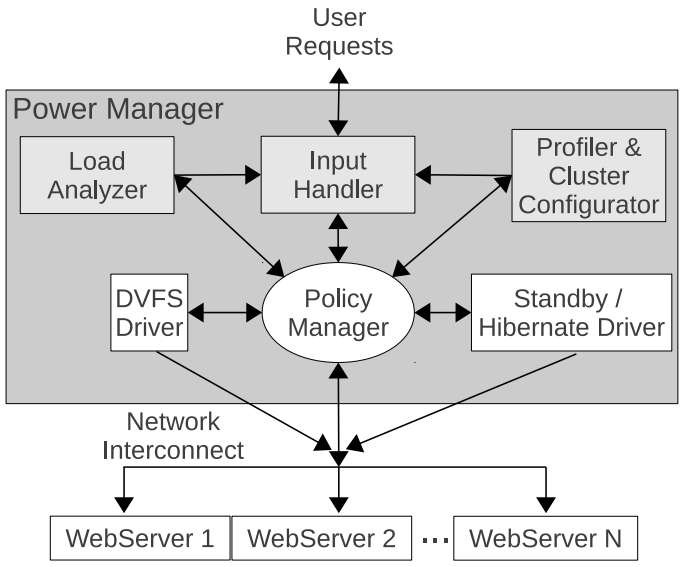


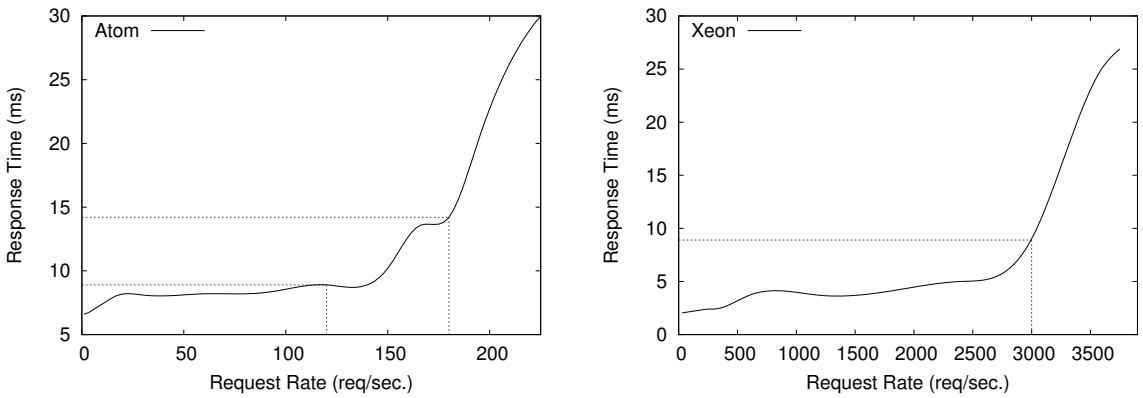
Figure 6.3 High-level system architecture of our power manager.

6.3 Power Manager for Heterogeneous Cluster

The input to the power manager is workload-independent power-performance characteristics of the different server types in the cluster, such as CPU model, idle power, P-states (CPU frequencies) and S-states (hibernate/suspend). We assume that the characteristics of the workload (e.g., average request rate, peak request rate and transition time) are available to the power manager in the form of traces or otherwise, as well as the SLA. The power manager profiles the workload and derives workload-specific power-performance characteristics for each server, such as power consumption, CPU utilization and response time for different request rates at different CPU frequencies. A lookup table is created for each server type for a given workload.

6.3.1 Relationship Between Response Time and Cluster Configuration

The response time knob is crucial in assessing the right composition of a cluster for a given workload. Figure 6.4 shows how the response time for the Dynamic Content Server workload varies with the request rate for Atom and Xeon (at peak frequency). The response time increases slowly with increasing request rate and then jumps to a high value (at which point



(a) Response time for Atom

(b) Response time for Xeon

Figure 6.4 Response time profiles for Atom and Xeon for Dynamic Content Server.

the error rate becomes non-zero). If we assume that 14 ms as response time is acceptable QoS for this particular workload, we find that a cluster composed entirely of Atom nodes would maximize the throughput per watt (as shown in Figure 6.2(b)). From Figure 6.2(a) it can be seen that around 16 Atom nodes can handle as much load as the Xeon node for this particular workload. Since a Xeon node would typically cost as much as a cluster of 10 Atom nodes or more and the throughput per watt difference is significant, it is safe to say that the TCO for a cluster of 16 Atom nodes would be less than that of a Xeon. The only reason to have Xeon nodes in the cluster at that point would be because the data center already had them before acquiring Atom nodes.

However, if the acceptable response time were 9 ms, the scale-out factor for Atom nodes would increase and so would the capital expenditure as well as power consumption (as suggested in [226]). Figure 6.2(c) incorporates the response time factor in the cost analysis. The Y-axis shows throughput per watt normalized by the response time. If the response time for a cluster of Atom nodes were to be matched with that of a Xeon, then for a request rate of more than 50 req/sec. for MediaWiki (and 600 req/sec. for Dynamic Content Server) a Xeon would be preferred over an Atom cluster, while for lower request rates, an Atom cluster would do better (in terms of power consumption), thus motivating a heterogeneous composition. In real life, the QoS would typically be defined by an SLA and the normalization factor would be a function of the response time. Our power manager incorporates the normalization function (response time for now) in deciding the optimal cluster composition/configuration.

In order to identify the optimal cluster configuration, the power manager runs the workload and finds the throughput per watt for different request rates for both Atom and Xeon (such that the number of violations/errors is zero). It notes the corresponding response times. It then divides the throughput per watt by the response time (which is assumed to be the normalization function) and stores the values in a table (one for each server type) indexed by the request rate. From the two tables, it estimates the point at which the normalized throughput per watt values for an Atom cluster would match that of a Xeon (i.e., the intersection point in Figure 6.1(c)). If such a point is found then the cluster configuration would be heterogeneous. The number of Atom nodes per Xeon would be equal to the number of Atom nodes needed to handle the request rate at the intersection point while keeping the response time within specified bounds (5 Atom nodes per Xeon for the Dynamic Content Server). If such a point is not found then the cluster configuration is homogeneous. If the normalized values for Atom are higher across the board, the cluster should be composed entirely of Atom nodes. If the normalized values for Xeon are higher then the cluster should be composed entirely of Xeon nodes.

6.3.2 Policy Manager

Once the cluster configuration is identified, the policy manager attempts to maximize throughput per watt by assigning P-states and S-states to the server nodes in the cluster for a given request rate such that the response time is within specified bounds and the error rate is zero. In the current implementation, the policy manager only works with the minimum and maximum frequency. That corresponds to 1 GHz. and 1.5 GHz. For Atom N550, and 1.6 GHz. and 2.4 GHz. for Xeon E5620. Only S3 sleep state (i.e., standby) is used. The following values are associated with each server type: P_{idle} (idle power), P_{minf} (peak power for the given workload at the lowest CPU frequency), P_{maxf} (peak power for the given workload for the highest CPU frequency), P_{stdby} (power consumed in standby mode), T_{minf} (max throughput handled at lowest CPU frequency for given response time), T_{maxf} (max throughput handled at highest CPU frequency for given response time). As mentioned before, these values are stored in a lookup table for each server type. For a given request rate, the policy manager tries to find the tuple $\{N_{idle}, N_{minf}, N_{maxf}, N_{stdby}\}$ for each server type (Atom and Xeon in this case) such that throughput per watt represented as:

$$\sum_{servertype} \frac{N_{minf}T_{minf} + N_{maxf}T_{maxf}}{N_{idle}P_{idle} + N_{minf}P_{minf} + N_{maxf}P_{peakf} + N_{stdby}P_{stdby}}$$

is maximized and

$$\sum_{servertype} ((N_{minf} + N_{maxf} + N_{idle}) \times T_{maxf}) - k \times reqRate$$

is minimized, subject to the constraints:

$$\sum_{servertype} ((N_{minf} + N_{maxf} + N_{idle}) \times T_{maxf}) - k \times reqRate \geq 0$$

and $N_{idle} + N_{minf} + N_{maxf} + N_{stdby} \leq N_{max}$.

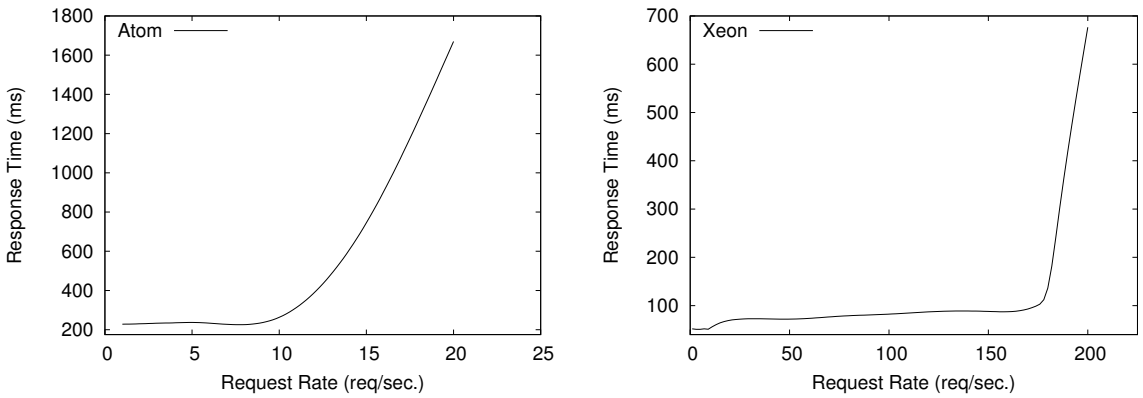
N_{max} is the maximum number of servers of each type available in the cluster. The factor k is determined by the nature of the workload. We set k as 2 in our experiments implying that the cluster is always ready for handling twice the current request rate. k is typically greater than 1 so as to handle unanticipated increase in load.

Under normal operation, the policy manager executes periodically and sets the power states of the servers appropriately. In order to handle load spikes, the policy manager constantly monitors the request rate and whenever it detects a sudden increase in request rate that persists for some time, the standby servers are alerted. The value of k (as mentioned before) is chosen in such a way that there are enough servers to handle the increase in load till the standby servers are ready for work. This design decision is based on the assumption that most load spikes take a minute or more to peak, which would give enough time for the standby servers to become operational. Once the spike subsides, the policy manager returns to its normal operation.

The power manager automatically derives a policy that uses both DVFS and standby mode. Under normal operation, the Xeon nodes are put to sleep, and the Atom nodes operate at low frequency (with the load distributed among the them intelligently). While a spike detector is used, no explicit workload prediction is carried out.

6.4 Evaluation

We experiment with two web server workloads: Dynamic Content Server (serving HTML pages from the local filesystem) and MediaWiki+MySQL. We compare the performance of



(a) Response time for Atom

(b) Response time for Xeon

Figure 6.5 Response time profiles for Atom and Xeon for MediaWiki.

Atom and Xeon nodes for the two applications to understand the power/performance trade-offs (as shown in Figure 6.1 and Figure 6.2) in order to identify the scale-out factor for Atom nodes. The acceptable response time and corresponding request rate handled for both workloads at different CPU frequencies is shown in Tables 6.1 and Table 6.2. The acceptable response times were set so that the throughput is maximized while keeping errors/violations=0 (this happens at the knee of the curve such as the one shown in Figure 6.5). For instance, at peak frequency Xeon node starts to drop requests and response time shoots up at about 176 req/sec.

In our evaluation, we focus on understanding the performance of different power management policies for a fixed cluster composition and workload. Since 1 Xeon can handle as much load as about 16-17 Atom machines for the two workloads, we fix the cluster size at 16 Atom netbooks + 1 Xeon server. We provision for twice the peak load, therefore peak is set at the capacity of 1 Xeon/16 Atoms (which is about 3000 req/sec. for Dynamic Content Server and around 160 req/sec. for MediaWiki). 1 Xeon and 32 Atom D510 servers are used for generating client side requests using *httperf* [227]. Power is measured using Watts Up Pro power meters. Before evaluating the power manager on a cluster of Atoms and Xeon, we try to understand the potential of DVFS on a cluster of Atom nodes alone.

Power State	Atom	Xeon
Idle	11.7	201.1
Standby (S-State)	2.4	24.1
Hibernate	1.0	21.1

Table 6.3 Power consumption (in Watts) for the Atom and Xeon servers under different power states.

6.4.1 Experimental Setup

The server cluster is composed of 16 Intel Atom N550 1.5 GHz. nodes each with two cores and 2 GB RAM, and 2 Intel Xeon E5620 2.4 GHz. nodes each with four cores and 48 GB RAM. Both Atom N550 and Xeon E5620 support DVFS, standby and hibernate modes. Atom takes 35 sec. and 90 sec. to wake up from the standby and hibernate modes respectively. Xeon takes 90 sec. and 120 sec. to wake up from the standby and hibernate modes respectively. Table 6.3 shows the power consumption for the Atom and Xeon servers under different power states.

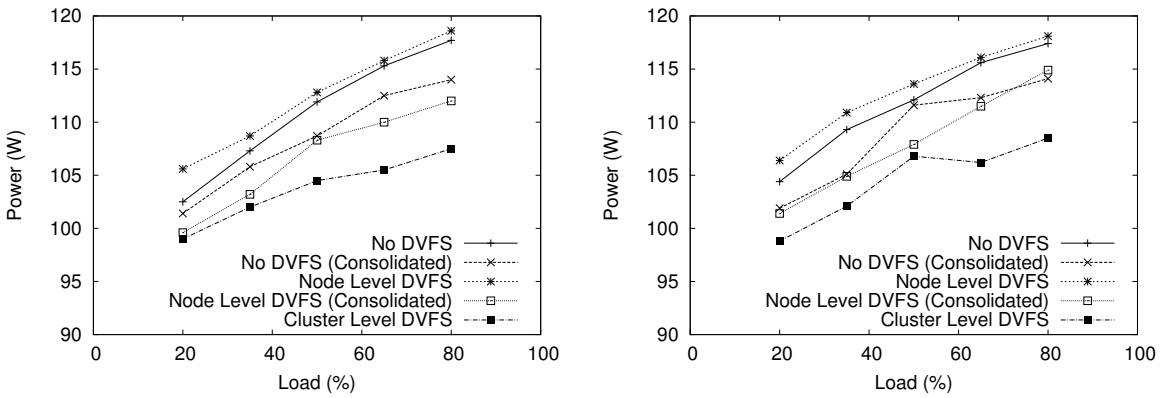
6.4.2 Impact of DVFS on Power Consumption of Atom Cluster

We now evaluate the effect of DVFS on Atom for web server workloads. For this experiment, we turn the Xeon server off completely and use only 8 Atom nodes. 100% load corresponds to the capacity of 8 Atom nodes i.e., 1500 req/sec. for Dynamic Content Server and 80 req/sec. for MediaWiki. Standby/hibernate modes are not used, therefore all the 8 Atoms are awake at all times. The load is gradually increased from 20-90%. We compare five different policies:

No DVFS: All the Atom nodes run at peak frequency (1.5 GHz.). No power management policy is used; the load is equally distributed among all the nodes.

No DVFS (Consolidated): All the nodes run at peak frequency. The load is consolidated and directed to the fewest number of nodes in the cluster.

Node Level DVFS: The default Linux policy governor (on-demand) is activated on all Atom nodes; each node is responsible for scaling its frequency based on CPU utilization. The load is equally distributed among the nodes.



(a) MediaWiki (b) Dynamic Content Server

Figure 6.6 Evaluation of DVFS on Atom cluster.

Node Level DVFS (Consolidated): This policy is similar to the previous one, however the input requests are consolidated and directed to the fewest number of nodes possible.

Cluster Level DVFS: All the Atom nodes are initialized to low frequency (1 GHz.). The load is balanced among the nodes. The frequency of a node is scaled up only when the capacity of the entire cluster at low frequency is saturated, which would happen when the load exceeds around 60%, since the maximum capacity of an Atom at low frequency is about 60% of the capacity at high frequency.

Figure 6.6 shows the power consumption of the cluster with respect to MediaWiki and Dynamic Content Server. Both applications show similar power consumption trends for the five policies. Interestingly enough, *Node Level DVFS* is the least power efficient among the five while *Cluster Level DVFS* comes out on top. Workload consolidation also helps: we observe an average improvement of 2.5% between *No DVFS* and *No DVFS (Consolidated)*, and 4.6% between *Node Level DVFS* and *Node Level DVFS (Consolidated)*. We observe an average gain of 6.5%, 4.3%, 7.6%, and 4% when using *Cluster Level DVFS* as compared to *No DVFS*, *No DVFS (Consolidated)*, *Node Level DVFS* and *Node Level DVFS (Consolidated)* respectively. Although the relative gains with *Cluster Level DVFS* are small (4-7%), they could translate to a few hundred thousand dollars to a corporation in energy savings per year.

6.4.3 Power Manager Performance Evaluation

We now evaluate our power manager on a cluster of 16 Atom nodes and 1 Xeon server. The power manager implements a meta-policy, which is to assign P-states and S-states to the cluster servers such that throughput per watt is maximized. In our design, the use of P-states and S-states is optional not mandatory. The policy generated by the power manager is compared against other well known policies:

No DVFS or Standby: All nodes run at peak frequency, no power management is carried out. The load is distributed among the nodes in the cluster.

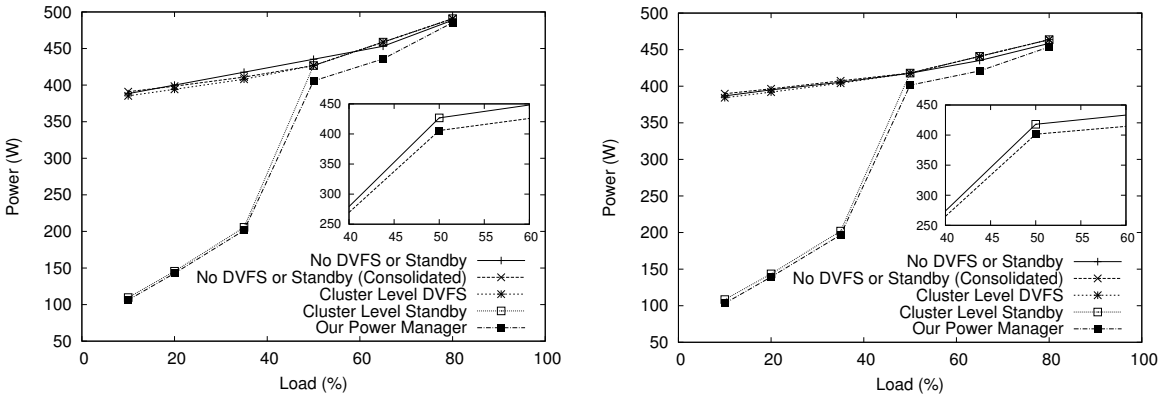
No DVFS or Standby (Consolidated): All the nodes run at peak frequency. The load is consolidated and directed to the fewest number of nodes in the cluster.

Cluster Level DVFS: Described in Section 6.4.2.

Cluster Level Standby: All the nodes run at peak frequency. The load is consolidated and directed to the fewest number of nodes in the cluster. The remaining nodes are put into standby mode.

As described in Section 6.3, in order to be able to sustain sudden load increases, when the request rate is r req/sec., the cluster should be prepared to handle $2 * r$ req/sec. (value of $k = 2$). The policy generated by the power manager uses a combination of P-states and S-states. At any given point in time, some of the nodes are in standby mode, some are idle, some are running at low frequency and others are running at peak frequency.

Figure 6.7 shows the power consumption of the policy generated by the power manager at equilibrium point. We find that when the load is low: 10-40%, the Xeon is in standby mode along with some of the Atom nodes, while others operate at either low or high frequency. When the load exceeds 50% all the standby nodes in the cluster are alerted. Note that the power consumption of the cluster suddenly goes up when the load exceeds 50%, which is due to the waking up of Xeon. As evident from Figure 6.7, the gains from standby are most significant when the load is $< 50\%$. Due to the scale of Figure 6.7, the power consumption curve of our policy manager seems to coincide with that of *Cluster Level Standby*. A closer look (as shown in the nested graph) reveals that there is a 3-4% net average gain with our policy manager when the load is $< 40\%$, which is due to DVFS. For higher load ($> 50\%$), the different policies tend to converge, and the gains from our policy manager (relative to Cluster Level Standby) become more pronounced (around 6%).



(a) MediaWiki (b) Dynamic Content Server

Figure 6.7 Power consumption with different power policies under increasing input load using heterogeneous cluster. DVFS+Standby gives an additional 3-4% savings as compared to Standby alone.

6.4.3.1 Workload Emulation

In order to evaluate the power manager in the presence of load spikes, we emulate a web server workload as shown in Figure 6.8. minutes, around 95-100% for about 6 minutes, 195-200% for about 9 minutes and the remaining 8 minutes are spent in between. Taking a cue from prior studies, we model the spikes such that it takes 90 sec. or more from the time the spike occurs till it reaches the peak. This gives enough time for the standby servers to wake up. Note that this workload pattern will not benefit our power manager, which yields higher energy savings when the load is between 50-80% (Figure 6.7). The workload emulation is meant to stress test the power manager.

We measure the total energy consumed by the cluster for MediaWiki application with the generated workload. Table 6.4 shows the energy savings obtained with our power manager and *Cluster Level Standby* as compared to the baseline: *No DVFS or Standby (Consolidated)*. The relative gain from the power manager with respect to the baseline is about 28.6%. The relative gain with respect to *Cluster Level Standby* is about 3.2%.

6.5 Chapter Summary

In this chapter, we explore strategies to improve the energy benefits of heterogeneous clusters by assigning DVFS (P-states) and low power sleep states (S-states) to heterogeneous

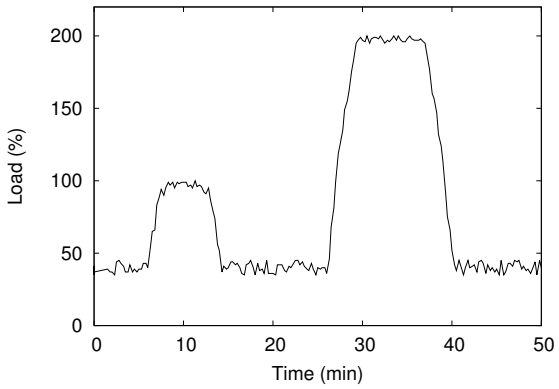


Figure 6.8 Generated workload for study the energy consumption with different power management schemes.

Power Management Scheme	Energy (kJ)	Energy Savings (%)
No DVFS/Standby (Consolidated)	413.5	0
Cluster Level Standby	304.9	26.2
Our Power Manager	295.2	28.6

Table 6.4 Energy consumption with different power management schemes for MediaWiki for the generated workload. Our power manager yields 3.2% improvement relative to Cluster Level Standby.

nodes. We design a cluster-level power manager that is able to automatically deduce the correct power states of the heterogeneous resources based on the current application load and the power profiles of the heterogeneous nodes. Our evaluation shows that compared to traditional power management policies, our cluster-level power manager significantly yields better throughput per watt for the studied enterprise scale applications, and maximizes the energy benefits of heterogeneous cluster.

Chapter 7

QoS-Aware Scheduling for Multi-Tenant Heterogeneous Clusters

Coprocessors, such as GPUs, are increasingly being deployed in clusters to process scientific and compute-intensive jobs. GPUs, in particular, are increasingly being used to accelerate non-graphical compute kernels, providing a $10 - 100\times$ performance boost for workloads such as linear system solvers, physical simulations, partial differential equations and flow visualizations [228–232]. At the same time, client-server applications which have traditionally been classified as compute- or data-intensive types now exhibit both characteristics simultaneously. Examples of such client-server applications are semantic search [233], video transcoding [234], financial option pricing [235] and visual search [236]. As in any client-server application, an important metric is *response time*, or the latency per request. For applications with enough parallelism within a single client request, latency per request can be improved by using GPUs. However, latency per request by itself is not enough. Multiple applications must be able to concurrently run and share a GPU-based heterogeneous cluster, i.e., the cluster must support multi-tenancy [237–239]. Further, client-server applications in practice experience varying rates of incoming client requests, sometimes even unpredictable load spikes. Thus, any practical heterogeneous cluster infrastructure must handle multi-tenancy and varying load, including load spikes, while delivering an acceptable response time for as many client requests as possible.

In order for a heterogeneous cluster to handle client-server applications with load spikes, a scheduler that enables dynamic sharing of heterogeneous resources is necessary. As an example, client requests of applications incurring load spikes should be processed by faster resources like the GPU, while requests of other applications could be deferred, or processed by slower resources. Without such a scheduler, decisions made for one application may

adversely affect another. For instance, sending one application's client request to the non multi-tasking GPU could block a more critical application.

In this chapter, we provide a scheduling solution for a multi-tenant GPU-based heterogeneous cluster to deliver acceptable response times (i.e., a response time that is less than or equal to the pre-specified response time) in the presence of load spikes. Response time is an important part of the system's Quality-of-Service (QoS), and is also the main concern of the client. We present a novel cluster-level scheduler, Symphony, that enables efficient sharing of heterogeneous cluster resources while delivering acceptable client request response times despite load spikes. Symphony manages client requests of different applications by assigning each request a priority based on the load and estimated processing time on different processing resources like the CPU and GPU. It then directs the highest priority application to issue requests to suitable processing resources within the cluster nodes. If necessary, the scheduler also directs applications to *consolidate* their requests (pack and issue multiple requests together to the same resource), and load-balances by directing client requests to specific cluster nodes.

7.1 System Architecture Overview

Figure 7.1 shows a high-level overview of the system. It consists of a GPU-based heterogeneous cluster hosting multiple client-server applications. The heterogeneous cluster has a cluster manager, which is a dedicated general-purpose multicore server node. It runs the cluster-level scheduler and application client interfaces. It manages a number of back-end servers, or *worker nodes*. The worker nodes contain heterogeneous computational resources comprising conventional multicores and CUDA-enabled GPUs. They expose their heterogeneity information to the cluster manager so that the manager can make appropriate decisions and schedule application user requests. All cluster nodes are interconnected using any standard interconnection network.

Each worker node hosts multiple applications concurrently on its resources. To isolate the performance of our framework, we assume that optimized architecture specific application code is available for the types of computational resources that we use, i.e., x86 CPU and GPU. That is, each of our applications has libraries containing optimized CPU and GPU implementations. These libraries are integrated within our middleware framework and used for scheduling, deriving CPU/GPU performance models and task (user request) dispatching.

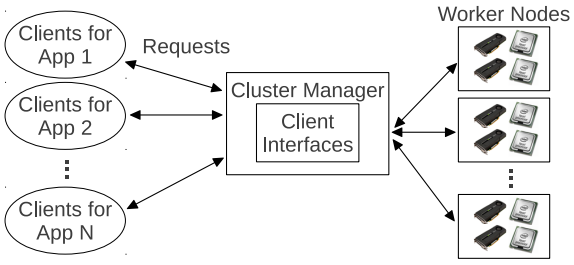


Figure 7.1 High-level system architecture of Symphony.

7.2 Application Characteristics and Interfaces

In this section, we describe our application characteristics, the cluster architecture, and define how applications interact with Symphony.

7.2.1 The Applications

We focus on applications that adhere to the client-server model and process remote client requests. Each application specifies an acceptable response time for its requests. We assume that all requests are of the same type, and only differ in size, e.g., semantic search processes text queries, but the queries can range in size from a single word to a large sentence. We make no assumptions about inter-dependency of client requests; after interfacing with Symphony, applications will still process requests in the order in which they were received.

All applications have a client interface and a server portion. The server portion, along with static application data, is mapped to specific cluster nodes, and is expected to be online and communicating with Symphony. When a client request arrives, it may be processed by one or more nodes where the application data is pre-mapped. Some applications may require all nodes to process each request, while others may just need any one node. Applications specify this information to Symphony, as we explain in Section 7.2.2.

Since we specifically target GPU-based heterogeneous clusters, we focus on applications whose request processing involves executing parallelizable compute kernels. We assume that applications already have optimized CPU and GPU implementations available in the form of runtime libraries with well-defined interfaces for such kernels. This enables Symphony to intercept calls to these kernels at runtime, and dispatch it to either CPU or GPU resources, as described later.

API	Description
void newAppRegistration(	**Application registers with scheduler.**
float `response_time`	Expected latency (ms) for each client request.
float `average_load`	Average number of requests expected every second.
int * `nodelist`	Possible cluster nodes on which a client request could be processed.
int `nodelist_size`	Size of above nodelist.
int `num_nodes`	Number of nodes necessary to process a client request.
int `consolidate`)	Number of requests that can be consolidated by application.
void newRequestNotification(	**Application notifies scheduler of the arrival of a new request.**
int `size`)	Size of data sent by request.
bool canIssueRequests(	**Application asks scheduler if requests can be issued.**
int * `num_reqs`	Number of consecutive requests that can be consolidated and issued.
int * `id`	Unique scheduler ID for this set of requests.
int * `nodes`	Which cluster nodes to use.
int * `resources`)	Which resources to use within each cluster node.
void requestComplete(	**Application informs scheduler that issued requests have completed processing.**
int `id`)	Scheduler ID pertaining to the completed requests.

Table 7.1 List of APIs exposed by Symphony.

Finally, some applications may have the ability to consolidate requests, i.e., pack and process multiple independent client requests together to achieve better throughput via increased parallelism. Symphony leverages this to drain pending requests faster.

7.2.2 Scheduler-Application Interface

We now define how client-server applications can communicate with a scheduler such as Symphony by making simple modifications. An application initially registers itself with Symphony and sends a notification each time it receives a client request. It then waits to receive the "go-ahead" from Symphony to process pending requests. Once requests have

completed processing, the application informs Symphony. Applications can use two threads to do this: one to notify the scheduler of new requests, and the other to ask if requests can be issued, and inform the scheduler of completion. This is not a major change since most client-server applications already do this to simultaneously fill buffers with incoming requests, and drain requests from the other end. The application modifications only require linking with the scheduler library and adding a few lines of code, with no reorganization or rewriting.

Table 7.1 provides the lists of the APIs exposed by Symphony. First, the application registers with the scheduler (`newAppRegistration()`) and specifies its expected response time for each client request (`latency`). The application also specifies the average number of client requests it expects to receive each second (`average_load`), the set of cluster nodes onto which its static data has been mapped (`nodelist`), and how many nodes each request will require for processing (`num_nodes`). For example, an application's data may have been mapped to 4 cluster nodes, but any of those 4 nodes can process a request. In this case, `nodelist` will contain names (or other descriptor) of the 4 nodes, and `num_nodes` will be 1. Finally, when an application registers with Symphony, it must also specify how many requests it can consolidate together (`consolidate`). For example, in the case of the Semantic Search, several user queries can be packed and executed simultaneously on a single worker node.

Applications notify Symphony of each new client request (`newRequestNotification()`) and specify the size of the request. In parallel, the application polls the scheduler to receive the go-ahead for processing pending requests (`issueRequests()`). Symphony tells the application how many requests to consolidate (`num_reqs`) and provides a unique identifier (`id`) for this set of requests. The application then processes the requests, and informs Symphony after they complete using `requestComplete()`.

7.3 Architecture of Symphony

Symphony is a request scheduler for multi-tenant client-server applications on heterogeneous clusters with a goal to deliver acceptable response times in the presence of load spikes. It combines application-specified parameters with its own inferences to make scheduling decisions. Symphony consists of cluster-level and node-level components. Figure 7.2 shows the manager node running the cluster-level component and client interfaces for the applications. The figure also shows the worker nodes running the node-level components. Both

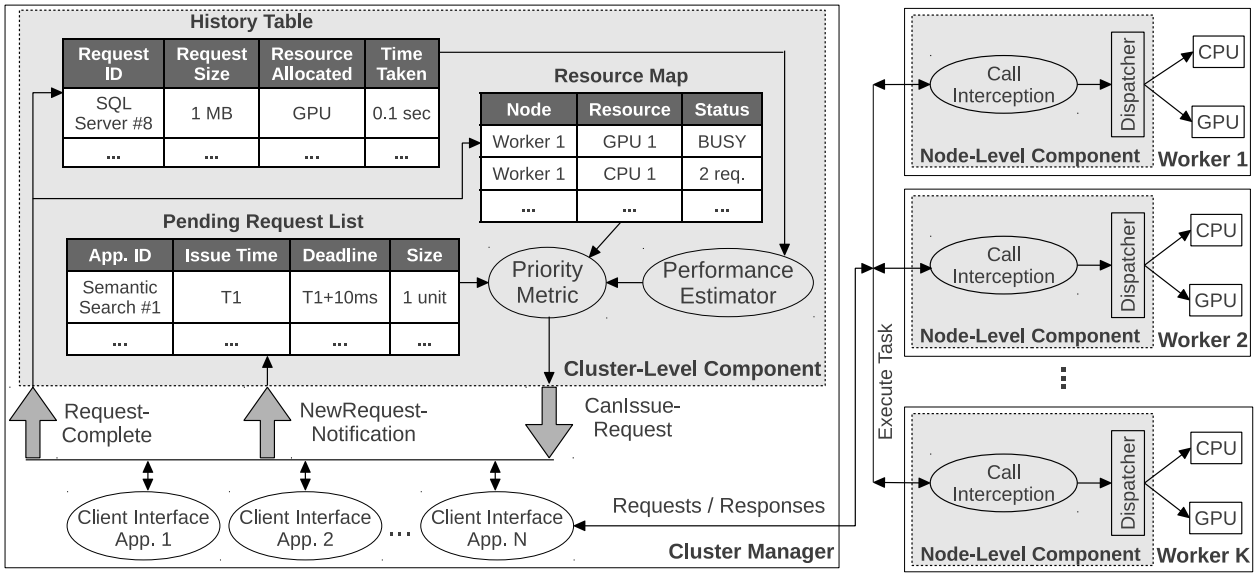


Figure 7.2 Architecture of Symphony.

components are implemented as user-space middleware.

7.3.1 Cluster-level Component of Symphony

This is the primary orchestrator in our system. Given client requests for different applications, it decides:

- which application should issue requests;
- how many requests should the application consolidate;
- to which cluster nodes should the requests be sent; and
- which resource (e.g., CPU or GPU) in the node should process the requests.

The architecture of the cluster component of Symphony consists of six portions as shown in Figure 7.2: (i) Pending Request List, (ii) Resource Map, (iii) History Table, (iv) Performance Estimator (v) Priority Metric Calculator and (vi) Load Balancer. We describe each of these below.

7.3.1.1 Pending Request List

Each application notifies Symphony upon the arrival of a client request. As shown in Figure 7.2, the scheduler stores certain information pertaining to pending requests, so that it can prioritize them and direct the applications to consolidate and dispatch the requests for processing. It does not store actual request data, but maintains for each request, the application that received the request, the time at which the request was received, the deadline by which the request should complete and the size of the request data.

7.3.1.2 Resource Map

Symphony monitors current cluster resource usage using a map of the CPU and GPU resources on each cluster node. For the CPU resource, it maintains a count of the number of requests being processed, while for the (non-multitasking) GPU, a BUSY/IDLE tag is maintained. This information is used to determine resource availability as well as to balance the load across the cluster. The resource map is updated each time the scheduler asks an application to issue requests (`issueRequests()`), and when an application informs the scheduler that it has completed requests (`requestComplete()`).

7.3.1.3 History Table and Performance Model

The history table stores details of recently completed requests of each application. Each entry of the history table contains a recently completed client request, resources that processed it, and the actual time taken to process the request. The history table is updated each time client requests complete (`requestComplete()`).

The information in the history table is used to build a simple linear performance model, the goal of which is to quickly estimate performance on the CPU or GPU so that the right requests can be issued with minimal response time failures. After collecting request sizes and corresponding execution times, we fit the data into a linear model to obtain CPU or GPU performance estimations based on request sizes. The model is dependent on the exact type of CPU or GPU; in our case we only have a single type of CPU and GPU, but if different generations of CPUs and GPUs exist, a model can be developed for each specific kind.

In addition to the dynamic performance model builder, existing analytical models can also be used to estimate the execution time of an application on available resources. Some analytical models such as [240] may require application and resource specific information at compile

time to accurately generate performance estimates. Although the performance model builder used by Symphony is simple, it requires no compile-time information to generate performance estimates.

7.3.1.4 Priority Metric

Symphony uses a priority metric to calculate the urgency of pending requests from the point of view of response time and overall load. Given N applications, where application A has n_A requests in the pending request list, the goal of the priority metric is to indicate (i) which of the N applications is most critical and therefore must issue its requests and (ii) which resource (e.g., CPU or GPU) should process the requests. Note that Symphony does *not* reorder requests within an application, but only across applications.

We assume that our heterogeneous cluster has r types of resources in each node, labeled R_1 through R_r. For example, if a node has 1 CPU and 1 GPU, r is 2. Furthermore, the application itself is responsible for actual request consolidation, but the scheduler indicates how many requests can be consolidated. To do this, the scheduler is aware of the maximum number of requests MAX_A that application A can consolidate (through `newAppRegistration()`). So if A is the most critical application, the scheduler directs it to consolidate the minimum of MAX_A or n_A requests.

If $DL_{k,A}$ is the deadline for request k of application A, CT the current time, and $EPT_{k,A,R}$ the estimated processing time of request k of application A on resource R, we define *slack* for request k of application A on resource R as:

$$slack_{k,A,R} = DL_{k,A} - (CT + EPT_{k,A,R}) \tag{7.1}$$

Initially, in the absence of historical information, $EPT_{k,A,R}$ is assumed to be zero. Resource R is either the CPU or GPU; if the system has different types of CPUs and GPUs, then each type is a resource since it would result in a different estimated processing time (EPT). A zero slack indicates the request must be issued immediately, and a negative slack indicates the request is overdue. Given the slack, we define urgency of request k of application A on resource R:

$$U_{k,A,R} = -slack^p \tag{7.2}$$

The above is a polynomial urgency functions, and we find that an exponent such as $p = 3$ provides good performance for our applications. We compare linear, polynomial and also exponential urgency functions in the results section. The above is the urgency of issuing a single request, and it increases polynomially as the slack nears zero. To account for load spikes, Symphony calculates the load L_A for each application A, using the average number of pending requests in the queue and the average number of requests expected every second ($navg_A$) specified at the time of application registration:

$$L_A = n_A / navg_A \tag{7.3}$$

We define the urgency of issuing the requests of application A on R as the product of the urgency of issuing the first pending request of A and the load of A:

$$U_{A,R} = \begin{cases} U_{1,A,R} \times L_A & , \; if \; R \; is \; available \\ \infty & , \; otherwise \end{cases} \tag{7.4}$$

We only consider the first pending request for each application because all application requests are processed in the order they are received, while requests across applications may be reordered. All pending requests of an application will therefore be less urgent than the first request.

The overall urgency U_A for issuing A's requests is the minimum urgency across all available resources R_i. If there are r different types of resources in each cluster node:

$$U_A = min_{i=1}^{r}(|U_{A,R_i}|) \tag{7.5}$$

Given the urgency for all applications, the scheduler will request application A to consolidate and issue q requests to resource R such that:

- Application A has the highest urgency among all applications;
- q is the minimum of MAX_A and n_A;
- Among all available resources, R is the resource when scheduled on which application A has minimum urgency.

Algorithm 7.1: Application selection algorithm of Symphony.

Input : $appList, reqList, resList, DL, EPT$
Output: app, q, R

for $A \in appList$ **do**
  $k = getEarliestRequest(A)$;
  $slack_{k,A,R} = calculateSlack(DL_{k,A}, EPT_{k,A,R})$;
  $U_{k,A,R} = calculateReqUrgency(slack_{k,A,R})$;
  $n_A = getAppReqCount(A)$;
  $navg_A = getAvgAppReqCount(A)$;
  $L_A = n_A / navg_A$;
  for $R \in resList$ **do**
    if $resAvailable(R)$ **then**
      $U_{A,R} = U_{k,A,R} \times L_A$;
    end
    else
      $U_{A,R} = \infty$;
    end
  end
  $U_A = getMinimum(U_{A,R}, resList)$;
end
/* Select application app to issue q requests to resource R */;
$app = getAppHighestUrgency()$;
$q = MIN(MAX_{app}, n_{app})$;
$R = getResWithLowestUrgency(app)$;

We note the following about the priority metric:

- If request falls behind in meeting its deadline, its urgency sharply increases (Equation 7.2).
- If an application experiences a load spike, its urgency sharply increases (Equation 7.4).
- Request processing is predicated on resource availability (Equation 7.4).
- For an application, the resource with the lowest urgency is the one with the best chance of achieving the deadline, and is therefore chosen (Equation 7.5).

Algorithm 7.1 shows an approach to implement the priority metric described above. It returns the application (app) with the highest urgency, the number of requests (r) that should

be consolidated together in the next dispatch, and the resources (R) on which the application should be executed. It is highly scalable since we do not compute the slack and urgency for every request in the pending request list, but only for the first MAX_A requests of every application. This keeps Symphony's overhead small, as we show in Section 7.4.

7.3.1.5 Load Balancer

As stated earlier, we assume static application data are pre-mapped to the cluster nodes. Client requests can be processed by a subset of these nodes, and the application tells the scheduler how many nodes are required to process a request (through `newAppRegistration()`). When the scheduler directs an application to issue requests, it provides a list of cluster nodes where the request can be processed by simply choosing the least loaded cluster node. The application is expected to issue its requests to these nodes and thus maintain overall load balancing.

7.3.2 Node-level Component

Besides the cluster-level scheduler that runs on the cluster manager node, separate node-level dispatchers [241, 242] run on each worker node. The node level dispatcher is responsible for receiving an issued request and directing it to the correct resource (CPU or GPU) as specified by the cluster-level scheduler. In order to do this, we assume that parallelizable kernels in the applications have both CPU and GPU implementations available as dynamically loadable libraries. The node-level dispatcher intercepts the call to the kernel, and at runtime directs it to either the CPU or GPU. For example, if processing a Semantic Search request requires a call to matrix multiplication, we assume that CPU and GPU library implementations are available for a specified function name, say `sgemm`. The node-level component intercepts `sgemm`, and looks for a directive from the cluster-level component. When the request was issued, the cluster component directly intimates the node-level component that `sgemm` in this instance of Semantic Search should be directed to, say the GPU.

7.4 Evaluation

In this section we describe our evaluation methodology and present results. We run four, full-fledged client-server applications concurrently on a high-end heterogeneous cluster with Intel Xeon CPUs and NVIDIA Fermi GPUs over a period of 24 hours. We subject the

applications to load spikes, where the duration and size of each spike are taken from published observations. Using our implementation of the scheduler as user-space middleware, we present the following results:

- *Priority Metric*: A comparison of Symphony's performance under different priority metrics and establish a "good" working metric for the following experiments.
- *Scheduler Performance Comparison*: A comparison of Symphony and baseline FCFS and EDF schedulers considering the number of dropped client requests, i.e., requests that do not meet response time constraints.
- *Efficient Cluster Sharing*: Empirical data showing that, compared to other schedulers, Symphony needs a smaller cluster to achieve the same performance.
- *Sensitivity to Load Spike Profile*: Unlike the baseline schedulers, Symphony performs well across a range of load spikes, i.e., spikes with varying height and width.
- *Scalability*: Data showing the running time of Symphony itself increases only marginally with increasing number of cluster nodes and applications.

For the first four set of results, the common metric of comparison is the number of client requests that do not meet response time constraints (QoS). We also call this "dropped requests".

7.4.1 Methodology

Our methodology consists of different sized heterogeneous clusters, with four real, end-to-end applications concurrently running on each cluster. We compare Symphony with two scheduling mechanisms, *First Come First Served* (FCFS) and *Earliest Deadline First* (EDF). In FCFS, client requests are processed in the same order as they arrive at the cluster manager. In EDF, the client request with the closest deadline is processed first. Both FCFS and EDF incorporate application placement and pre-mapped data while making scheduling decisions. Furthermore, FCFS and EDF consider GPUs as well as CPUs while scheduling application requests on the available nodes. However, GPUs are preferred: requests are processed on the CPUs only if all GPUs are busy.

We now describe the applications, cluster configurations and spike introduction mechanisms.

Application	Description	Response Time
Semantic Search	Supervised Semantic Indexing [233] (SSI) matching to search large document databases. It searches the indexed documents for the user queries and ranks the results based on their semantic similarity to the given queries.	5 msec/query
Video Transcoding	An implementation [234] of x264 that converts the video streams into the H.264/MPEG-4 AVC format. Each cluster node executes an instance of Video Transcoding application to encode the given video stream.	500 msec/MB
SQL Server	An implementation [66] of a subset of SQLite commands processor. Each worker executes an instance of SQL Server hosting the same database.	150 msec/query
Option Pricing	An implementation [243] of Black-Scholes financial model to compute the evolution of future option prices. Each worker hosts an instance of Option Pricing and provides the option prices for user queries.	800 msec/query

Table 7.2 Enterprise applications with execution resources and performance criteria.

7.4.1.1 Enterprise Applications

Emerging cluster computing workloads consist of a mix of short- and long-running jobs. We have selected four representative applications from different domains covering the spectrum of latency- and throughput-intensive workloads. We choose Semantic Search, SQL Server, Video Transcoding and Option Pricing as our representative workload set. Table 7.2 provides the brief description of each application along with their performance requirements, and Table 7.3 describes the data layout for each application. Some of these applications, i.e., Semantic Search, and SQL Server, execute short-running tasks, while others execute long-running jobs. The table also describes the application's static data layout. Based on data layout, a client request can be processed on a subset of worker nodes (e.g., Semantic Search), or on any available worker node (e.g., Video Transcoding, SQL Server, Option Pricing) of the heterogeneous cluster.

Application	Data Layout	Data Size
Semantic Search	Document repositories distributed across worker nodes so that each document is available on at least two worker nodes.	2 million documents
Video Transcoding	Input video data accessible at each worker node through Network File System (NFS) [244].	$4 - 45$ MB
SQL Server	Database replicated on all the worker nodes. Any worker node can process the given query.	512 MB
Option Pricing	Input data accessible at each worker node through NFS.	400 MB

Table 7.3 Enterprise applications with data layout and data size.

7.4.1.2 Cluster Configurations

Our cluster consists of seven high-end worker nodes, and a manager node. Each worker node has two Intel Xeon E5620 processors (2.4 GHz each), with QPI and 48 GB main memory. A worker node also has two 1.3 GHz NVIDIA Fermi C2050 GPUs with 3 GB internal memory, connected as coprocessors on PCI-Express slots. The worker nodes are interconnected together, and with the manager using gigabit ethernet.

7.4.1.3 Spike Introduction Mechanism

According to published observations, typical spike durations vary from $10 - 30$ minutes, while the peak of the spikes can be as high as $1.5\times$ of the normal load [224, 225, 245, 246]. We introduce random load spikes with duration and heights in this range, but extend our evaluation to a broader range of spikes. Our spike introduction mechanism injects spikes at random time points for each application, independent of the other applications running at the same time.

7.4.2 Priority Metric

Before we present results with Symphony, we explore the priority metric used in order to empirically establish a good enough heuristic for the rest of the experiments. Specifically, we replace the polynomial urgency function $-slack^3$ from Equation 7.2, with exponential and linear functions, and compare the final performance of the system in terms of the number of client requests dropped. The polynomial function ($-slack^3$) was replaced with exponential

Spike Height	Polynomial ($-slack^3$)	Linear ($-slack$)
Spike Width: 5 *minutes*		
1.25	0.02	0.19
1.5	0.05	0.32
2.0	0.8	3.31
Spike Width: 15 *minutes*		
1.25	0.02	0.16
1.5	0.04	0.29
2.0	0.81	3.16

Table 7.4 Average percentage of requests per minute not meeting the specified QoS under different slack functions.

(2^{-slack}) and linear ($-slack$) functions. We find out that the average percentage of dropped requests are similar for the exponential and polynomial urgency functions, while the simple linear urgency function underperforms. Table 7.4 compares polynomial and linear functions for different load spike heights and widths, where the height of the spike is the ratio between the spike and normal application load, and the width is the duration of the spike in minutes. The data shows that using the linear urgency function table incurs far more requests dropped than the polynomial (or exponential) urgency functions.

This is expected since the urgency function defines how responsive the system is to spikes. In a sense, the urgency function has to "follow" a spike closely; the faster a spike rises, the quicker the urgency of the those pending requests should become. Our data establishes that in general polynomial or exponential urgency functions follow random load spikes well. We thus use the polynomial urgency function for the rest of the experiments in this section.

7.4.3 Scheduler Performance Comparison

We now compare the performance of Symphony versus baseline FCFS and EDF schedulers using a 5-node heterogeneous cluster. The metric for the comparison is the number of client requests that do not meet response time constraints.

7.4.3.1 Low Application Load

We first run all four applications concurrently under low load conditions. Not surprisingly, we find that at low loads, all scheduling schemes perform well, i.e., the fraction of dropped requests is under 0.01% for FCFS, EDF and Symphony. For instance, for EDF and Sym-

Application	No. of Requests Processed	Average Desired Latency/Request
Semantic Search	17.2 M	5 msec.
Video Transcoding	52 K	45 sec.
SQL Server	576 K	150 msec.
Option Pricing	108 K	800 msec.

Table 7.5 Number of requests processed with average desired latency.

phony the maximum fraction of dropped requests at any instance is 0.012% for and 0.011% for Video Transcoding, respectively. Thus EDF and Symphony exhibit similar performance for our applications under low application load.

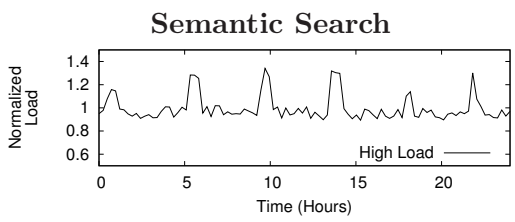
7.4.3.2 Realistic Application Load (with Spikes)

Next we study the performance under varying load conditions using the realistic spike introduction model of Section 7.4.1.3. Figure 7.3 shows the results. For FCFS, the maximum observed percentage of dropped requests is around 7%, which occurs for Video Transcoding. On average FCFS drops 3.3% of requests across all applications over 24 hours. For EDF and Symphony, the maximum fraction of dropped requests is 3.5% and under 2%, respectively, occurring for Video Transcoding and Option Pricing. Across all four applications over 24 hours, EDF (which is better than FCFS) drops close to 2% of requests on the average, while Symphony drops under 1%. This shows that Symphony is effective in handling load spikes when given a fixed number of heterogeneous cluster nodes. This is inline with expectations since FCFS does not consider request deadlines at all, while EDF, although it prefers requests that are closest to missing deadlines, does not consider estimated request processing time.

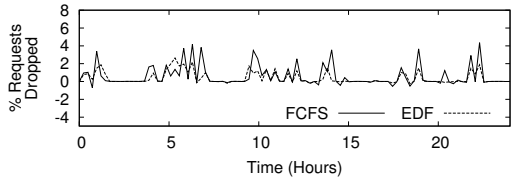
7.4.4 Efficient Cluster Sharing

We now study how Symphony performs with limited resources. Specifically, we reduce the cluster size to 3 worker nodes, and experimentally determine the number of dropped requests for all three scheduling schemes. Then we gradually increase the cluster size, and find that the baseline scheduling schemes need a cluster with more worker nodes to provide the same fraction of requests dropped.

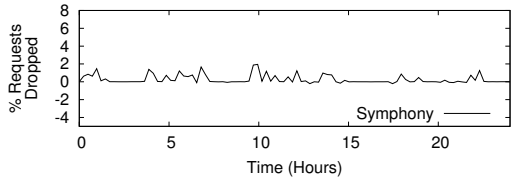
Table 7.5 and Table 7.6 show the results using the realistic load profile from Figure 7.3(a).



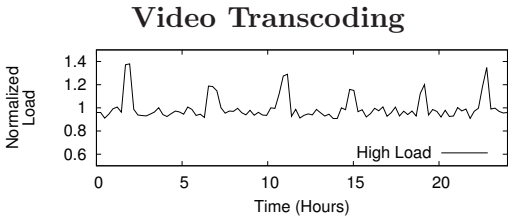
(a) Application load.



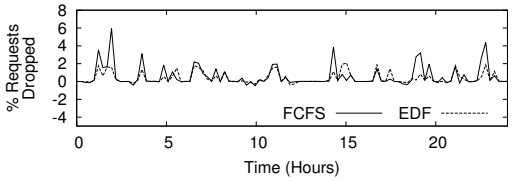
(b) Performance with FCFS and EDF.



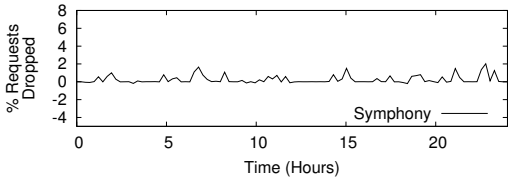
(c) Performance with Symphony.



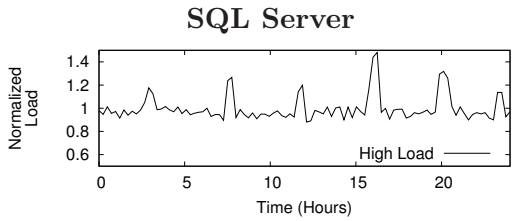
(a) Application load.



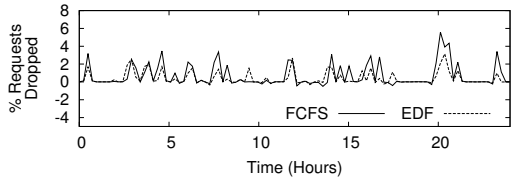
(b) Performance with FCFS and EDF.



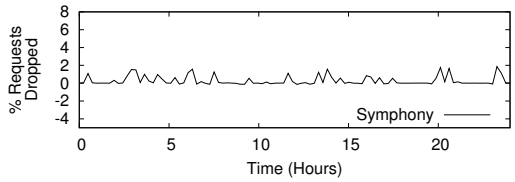
(c) Performance with Symphony.



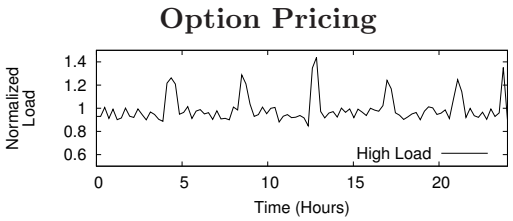
(a) Application load.



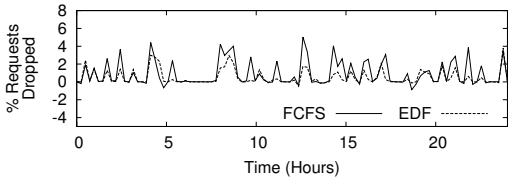
(b) Performance with FCFS and EDF.



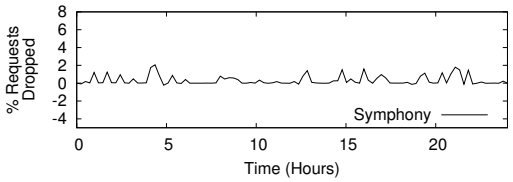
(c) Performance with Symphony.



(a) Application load.



(b) Performance with FCFS and EDF.



(c) Performance with Symphony.

Figure 7.3 Requests missing QoS under spike load conditions for the four concurrently running applications.

Application	3-Node Cluster			7-Node Cluster		
	FCFS	EDF	Symphony	FCFS	EDF	Symphony
Semantic Search	24.1	20.2	3.0	2.9	1.2	0.05
Video Transcoding	27.2	21.6	3.3	3.2	1.3	0.05
SQL Server	21.8	14.7	2.8	2.6	1.1	0.04
Option Pricing	25.9	20.9	3.2	3.1	1.4	0.05

Table 7.6 Percentage of requests dropped under different cluster configurations for 24 hour period.

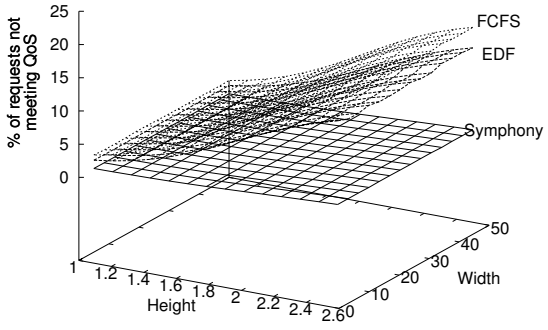


Figure 7.4 Percentage of requests per minute not meeting QoS with increasing spike height (normalized to average load) and width (minute).

With thousands to millions of requests and a 3-node cluster, FCFS and EDF end up dropping 14 to 27% of requests, while Symphony drops around 3%. FCFS and EDF attain the same level of performance as Symphony, i.e., 3% dropped requests, when provisioned with 7 cluster nodes. At that cluster size however, and the same load profile, Symphony drops only around 0.05% of requests. Therefore, with its load- and heterogeneity-sensitive scheduling, Symphony shares cluster resources more efficiently under varying load conditions, leading to better utilization of the cluster.

7.4.5 Sensitivity to Load Profile

To study how Symphony performs under different types of load spikes, we introduced spikes of varying height and width and measured the average percentage of dropped requests on the 7-node heterogeneous cluster. Figure 7.4 shows the percentage of dropped requests versus the height and width of the spikes. The height of spike is relative to its height at low load,

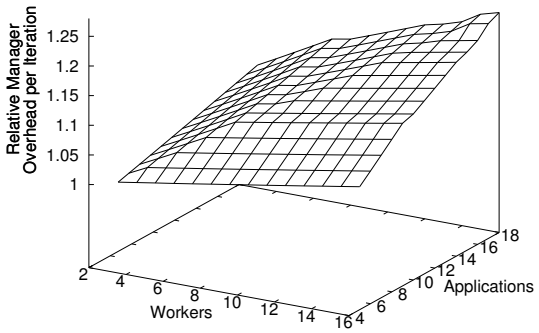


Figure 7.5 Scalability with increasing workers and applications.

while the width is its duration in minutes. The performance of FCFS and EDF deteriorates sharply with spike height. The maximum percentage of dropped requests is 20.5%, 18.0% and 2.15% for FCFS, EDF and Symphony respectively. Across all points, the average percentage of dropped requests per minute is 9.2%, 7.2% and 0.7% for FCFS, EDF and Symphony respectively. This shows that Symphony is not very sensitive to the type of load spike, and can handle a broader range of spikes that are outside previously published observations.

7.4.6 Scheduler Scalability

A concern with the single-manager design is the overhead due to introduction of additional middleware layers, as this could lead to performance bottlenecks when operating at scale. To this end, we observed how the performance of Symphony scales with the number of worker nodes and with the number of applications. We increase the number of worker nodes from 3 to 16 and the number of applications from 4 to 18. To run more than four applications, we replicate our existing applications.

Figure 7.5 shows the overhead of Symphony, i.e., the time taken by the scheduler itself, normalized to the overhead for a 3 node heterogeneous cluster executing four applications. We see that it scales nearly linearly (with a small slope) as the number of worker nodes and applications increase. The four corners of the 3-D plot represent: (i) 3-node cluster with four applications; (ii) 3-node cluster with 18 applications; (iii) 16-node cluster with four applications; and (iv) 16-node cluster with 18 applications. The scheduler takes longest

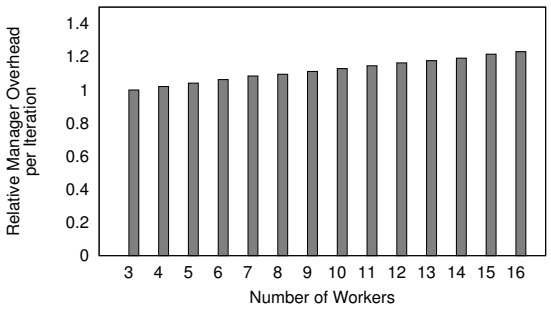


Figure 7.6 Symphony overhead with increasing workers (diagonal from Figure 7.5).

when both the number of cluster nodes as well as the number of concurrent applications are high. However, the running time of Symphony at 16 cluster nodes with 18 concurrent applications is only 22% larger than the base case of 3 cluster nodes with four applications. The marginal increase and Symphony's scalability is apparent from Figure 7.6 (the diagonal from Figure 7.5), which shows the overhead versus cluster size where a cluster with m nodes hosts $m + 1$ applications.

7.5 Chapter Summary

In this chapter, we enhance our framework by designing Symphony, a cluster-level heterogeneous resource scheduler, that takes into consideration the required performance requirements of individual applications while making the scheduling decisions. Symphony enables efficient sharing of heterogeneous resources between concurrently executing multiple applications with strict performance requirements in a multi-tenant environment. It computes the priority of issuing the user requests for each application, and reorders them based on their performance requirements and associated deadlines. Our evaluation using enterprise-scale applications under varying load conditions shows that compared to the traditional schedulers, specifically FCFS and EDF, Symphony significantly reduce the number of user requests that do not meet the desired performance requirements. Furthermore, Symphony is scalable and provides efficient sharing of heterogeneous resources with better cluster utilization compared to the traditional resource scheduling mechanisms.

Chapter 8

Conclusions

This dissertation presents the design of an adaptive framework for managing accelerator-based heterogeneous many-core clusters for data-intensive computing. The framework addresses the challenges of designing intuitive programming models, and developing efficient resource scheduling mechanisms for large-scale asymmetric clusters comprising of heterogeneous accelerator-based compute nodes with varying I/O and compute capabilities. The use of the proposed framework reduces the time-to-solution for modern applications, and improves the performance of the asymmetric clusters by avoiding I/O and computational stalls at the heterogeneous resources. It schedules the applications on the available resources based on the desired performance requirements of individual application and the computational capabilities of asymmetric nodes. Furthermore, the framework provides energy- and QoS-aware resource scheduling mechanisms to yield maximum benefits from the available heterogeneous resources.

Our extended MapReduce-based programming model, CellMR, provides an intuitive programming framework for heterogeneous clusters of asymmetric multicores, such as IBM Cell, and computational accelerators, such as programmable GPUs, coupled with conventional general-purpose multicores in a distributed setting. CellMR relies on a streaming approach and dynamic data distribution techniques to implement an architecture-agnostic, yet scalable programming model for asymmetric distributed clusters, featuring accelerators at their compute nodes. It preserves the simple, portable, and fault-tolerant programming interface of MapReduce, while exploiting multiple interconnected computational accelerators for HPC. The framework also provides dynamic schemes for memory management and work allocation, so as to best adapt work and data distribution to the relative computation density of the application and to the variability of computational and storage capacities across asymmetric components. The proposed dynamic data distribution schemes enable higher

performance and better utilization of the available memory resources, which in turn helps economizing on capacity planning for large cluster installations.

We also investigate an advanced software engineering approach to building software for the accelerator-based heterogeneous clusters. Our approach adopts the well-established layered software architecture implemented as mixin-layers. Applying this approach not only achieves the required high performance, but also enables the reuse of the majority of the code, when types, number, or hierarchy of accelerators is changed. The evaluation of mixin-layers based reusable software components indicates that our approach enables effective reuse at the software component granularity, which can reduce the time-to-solution metrics for creating accelerator-based heterogeneous clusters. Currently, leveraging these resources for HPC requires the effort and expertise commensurate to that of a seasoned computer scientist. However, the ability to build software for these resources from reusable software components has the potential to lower the barrier-to-entry for accelerator-based clusters, making them accessible to researchers from a myriad of scientific and engineering fields.

We research prefetching techniques for supporting data-intensive applications on the IBM Cell multiprocessor, and observe that the current operating system facilities for performing I/O directly on accelerator cores (SPEs) are limited, and do not provide judicious use of the available resources. A particular concern is that currently, I/O on SPEs is redirected to the PPE, hence creating a central bottleneck. We present an asynchronous prefetching-based approach that partially breaks up this bottleneck, utilizes decentralized DMA to achieve (22.2%) better performance compared to the case where all I/O is handled at the SPE.

We also present system design alternatives and capabilities-aware task scheduling for large-scale data processing on accelerator-based distributed systems. We enhanced CellMR with runtime support for utilizing multiple types of computational accelerators via runtime workload adaptation and for adaptively mapping MapReduce workloads to accelerators in virtualized execution environments where applications are distributed in space and time on shared resources. We find that adaptively matching application execution properties to the computational capabilities of accelerators significantly improves (up to 26.9%) system performance compared to the static workload distribution techniques.

Our energy-aware resource management module provides a power manager that finds the optimal cluster configuration for a given workload and then goes on to maximize the work done per watt by dynamically assigning P-states and S-states to the cluster nodes based

on the current application request rate. We find that the composition of a cluster, i.e., the decision of whether or not a cluster should be heterogeneous and if so what the optimal configuration should be, depends on the characteristics of the workload and the capabilities of the heterogeneous nodes. We automate the process of finding the optimal cluster configurations as part of our power manager that generates the policy that yields maximum energy savings. The generated policy suggests the appropriate use of cluster-level DVFS and standby mode. Although the relative gains from DVFS were found to be small (3-6% in our setup), it might translate to a few hundred thousand dollars per year for a large-scale cluster. As compared to the baseline, our generated policy shows significant energy savings (28.6%) for the generated workload.

Finally, to support executing multiple concurrent applications with strict performance requirements on a heterogeneous cluster in a typical cloud computing environment, we present a novel priority based scheduler, called Symphony, that is characterized by three key attributes. First, it continuously monitors the load on each application. Second, it collects past performance data and dynamically builds simple performance models of the available heterogeneous processing resources. Third, it computes a priority for pending requests based not only on the user-specified parameters about the application but also on the information inferred from its performance models. Symphony is largely immune to load spikes and maintains acceptable response times despite fairly large load variations, and incurs $(2-20\times)$ fewer requests that do not meet the performance requirements compared with other schedulers. Furthermore, our approach reveals that a QoS-aware resource scheduler can be effectively used to significantly reduce the number of compute nodes in a heterogeneous cluster without sacrificing the system performance. Specifically, in order to match the performance of Symphony, other schedulers need $(2\times)$ more cluster nodes.

8.1 Future Research

Heterogeneous computing is a norm and large-scale heterogeneous systems are the future. The framework presented in this dissertation addresses the challenges of programming and managing many-core heterogeneous clusters comprising commodity accelerators. However, there remain several barriers to the efficient use of such clusters by the community at large. In the following, we discuss few research projects that are natural extensions of the framework presented in this dissertation.

8.1.1 Generic Programming Models for Heterogeneous Systems

Fully leveraging the parallelism opportunities offered by heterogeneous architectures poses new challenges which brings into question conventional programming models and abstractions for designing large-scale systems. In this dissertation, we have explored how high-level programming languages can be used efficiently to develop reusable components and middlewares for heterogeneous clusters. However, automatic parallelization of sequential programs to exploit the inherent parallelism offered by the next generation architectures remains an open issue. We plan to investigate innovative compiler techniques to automatically parallelize and optimize the code generation for heterogeneous systems. Furthermore, we intend to leverage the lessons learned from developing an extended MapReduce model to design more generic programming models for accelerator-based distributed systems.

8.1.2 Inter-Application Interference-Aware Resource Scheduling

The framework presented in this dissertation provides power-aware and QoS-aware resource scheduling mechanisms for heterogeneous clusters. However, in a multi-tenant accelerator-based heterogeneous cluster, it is critical to schedule concurrent applications that have minimum contentions for the same accelerator resources on the same compute node to improve the overall system performance. If the applications having similar resource access patterns are scheduled on the same compute node, then while one application is utilizing the accelerator, the others would stall and wait for their share in time and space for the occupied accelerator resources. This would result in reduced system throughput and may violate the essential QoS requirements for critical applications. We plan to extend the resource scheduling mechanisms presented in this work, and design an inter-application interference-aware resource scheduling mechanism that executes the applications with contentions for the same accelerator resources on separate compute nodes to increase the concurrency and improve the utilization of heterogeneous cluster.

8.1.3 Virtualizing Computational Accelerators

The use of emerging accelerators and asymmetric multicores in data centers is critical for providing high-performance with reduced setup and energy costs. Data centers typically provide virtual machine containers that host dedicated applications for individual users in a secure environment. A critical issue in enabling the use of powerful accelerators and asym-

metric multicores for enterprise computing is the lack of virtualization support for these architectures. We intend to investigate how emerging accelerators and asymmetric multicores can be used efficiently in a virtualized setup and develop new computational models and runtime frameworks to enable the use of these architectures in secure and cost-efficient environments.

8.1.4 Supporting Operating System Operations

Emerging heterogeneous and asymmetric parallel architectures pose new research venues for improving the performance of traditional operating systems while providing challenging opportunities to design next generation operating systems. The current state of the art does not support executing operating system services on these massively parallel architectures, hence limiting their use to improve the performance of user applications only. We intend to investigate the design of next generation operating systems services that can leverage the degree of parallelism, execution models and memory bandwidth offered by asymmetric multicore and many-core processors to improve the overall system performance.

Bibliography

[1] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation - a performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[2] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[3] AMD. The AMD Fusion Family of APUs, 2011. `http://www.fusion.amd.com/`.

[4] NVIDIA Corporation. NVIDIA CUDA Programming Guide. November 2007.

[5] Jason Cross. A Dramatic Leap ForwardGeForce 8800 GT, Oct 2007. `http://www.extremetech.com/article2/0,1697,2209197,00.asp`.

[6] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[7] Intel. Single-chip Cloud Computer, 2010. `http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC-Overview.pdf`.

[8] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, June 2005.

[9] M. Hill and M. Marty. Amdahl's Law in the Multi-core Era. Technical Report 1593, Department of Computer Sciences, University of Wisconsin-Madison, March 2007.

[10] M. Pericàs, A. Cristal, F. Cazorla, R. González, D. Jiménez, and M. Valero. A Flexible Heterogeneous Multi-core Architecture. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–24, September 2007.

[11] Kumar R., K. Farkas, N. Jouppi, P. Ranganathan, and D. M. Tullsen. Processor Power Reduction via Single-ISA Heterogeneous Multi-core Architectures. *Computer Architecture Letters*, 2, 2003.

[12] Kumar R., D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[13] H. Wong, A. Bracy, E. Schuchman, T. Aamodt, J. Collins, P. Wang, G. Chinya, A.Khandelwal Groen, H. Jiang, and H. Wang. Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor. In *Proc. of the 17th IEEE International Conference on Parallel Architectures and Compilation Techniques*, Toronto, Canada, October 2008.

[14] AMD. The Industry-Changing Impact of Accelerated Computing. 2008.

[15] David Bader and Virat Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In *Proc. of the 14th IEEE International Conference on High Performance Computing (HiPC), Lecture Notes in Computer Science 4873*, December 2007.

[16] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. RAxML-CELL: Parallel Phylogenetic Tree Construction on the Cell Broadband Engine. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, March 2007.

[17] G. Buehrer and S. Parthasarathy. The Potential of the Cell Broadband Engine for Data Mining. Technical Report TR-2007-22, Department of Computer Science and Engineering, Ohio State University, 2007.

[18] Bugra Gedik, Rajesh Bordawekar, and Philip S. Yu. Cellsort: High performance sorting on the cell processor. In *Proc. of the 33rd Very Large Databases Conference*, pages 1286–1207, 2007.

[19] Sandor Heman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized Data Processing on the Cell Broadband Engine. In *Proc. of the Third International Workshop on Data Management on New Hardware*, June 2007.

[20] Fabrizio Petrini, Gordon Fossum, Juan Fernández, Ana Lucia Varbanescu, Michael Kistler, and Michael Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[21] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: The architecture and performance of Roadrunner. In *Proc. Supercomputing*, 2008.

[22] ClearSpeed Technology. *ClearSpeed whitepaper: CSX processor architecture*, 2007.

[23] Isaac Gelado, Javier Cabezas, Nacho Navarro, John E. Stone, Sanjay J. Patel, and Wen mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In James C. Hoe and Vikram S. Adve, editors, *ASPLOS*, pages 347–358. ACM, 2010.

[24] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC'04, Washington, DC, USA, 2004. IEEE Computer Society.

[25] James C. Phillips, John E. Stone, and Klaus Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC'08, pages 8:1–8:9, Piscataway, NJ, USA, 2008. IEEE Press.

[26] Duc Vianney, Gad Haber, Andre Heilper, and Marcel Zalmanovici. Performance analysis and visualization tools for cell/b.e. multicore environment. In *IFMT'08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, pages 1–12, New York, NY, USA, 2008. ACM.

[27] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. Cuda-lite: Reducing gpu programming complexity. pages 1–15, 2008.

[28] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 287–296, Seattle, WA, March 2008.

[29] Perry Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multi-threaded System. In *Proc. of the 2007 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 156–166, San Diego, CA, 2007.

[30] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. Memory - cellss: a programming model for the cell be architecture. In *Proc. of Supercomputing'2006*, page 86, 2006.

[31] Marc de Kruijf and Karthikeyan Sankaralingam. MapReduce for the Cell B.E. Architecture. Technical Report TR1625, Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, November 2007.

[32] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally,

and Pat Hanrahan. Memory - sequoia: programming the memory hierarchy. In *Proc. of Supercomputing'2006*, page 83, 2006.

[33] Message Passing Interface Forum. MPI2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):299, 1998.

[34] K. Feind. Shared memory access (SHMEM) routines. In *Cray User Group, Inc.*, 1995.

[35] K. Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report EECS-TR-2006-183, Electrical Engineering and Computer Science Division, University of California, Berkeley, December 2006.

[36] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc., Nov 2010. `http://aws.amazon.com/ec2/`.

[37] Dilip Kandlur. Storage challenges for petascale systems. In *Fifth Intelligent Storage Workshop*, May 2007. `http://www.dtc.umn.edu/disc/resources/KandlurISW5.pdf`.

[38] Bill Allcock, Ian Foster, Veronika Nefedova, Ann Chervenak, Ewa Deelman, Carl Kesselman, Jason Lee, Alex Sim, Arie Shoshani, Bob Drach, and Dean Williams. High-performance remote access to climate simulation data: a challenge problem for data grid technologies. In *Proc. 2001 ACM/IEEE conference on Supercomputing*, pages 46–46, Denver, CO, Nov. 2001.

[39] Paul Krueger. High performance computing storage challenges. In *Keynote Talk. Fifth Intelligent Storage Workshop*, May 2007. `http://www.dtc.umn.edu/disc/resources/KruegerISW5.pdf`.

[40] Catherine H. Crawford, Paul Henning, Michael Kistler, and Cornell Wright. Accelerating computing with the cell broadband engine processor. In *CF'08: Proceedings of the 2008 conference on Computing frontiers*, pages 3–12, New York, NY, USA, 2008. ACM.

[41] M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, and Dimitrios S. Nikolopoulos. Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[42] M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, and Dimitrios S. Nikolopoulos. Supporting mapreduce on large-scale asymmetric multi-core clusters. *ACM SIGOPS Operating Systems Review*, 43(2):25–34, 2009.

[43] M. Mustafa Rafique, Ali R. Butt, and Dimitrios S. Nikolopoulos. Designing accelerator-based distributed systems for high performance. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID'10, pages 165–174, Washington, DC, USA, 2010. IEEE Computer Society.

[44] M. Mustafa Rafique, Ali R. Butt, and Eli Tilevich. Reusable software components for accelerator-based clusters. *Journal of Systems and Software*, 84:1071–1081, July 2011.

[45] M. Mustafa Rafique, Ali R. Butt, and Dimitrios S. Nikolopoulos. DMA-based prefetching for I/O-intensive workloads on the cell architecture. In *CF'08: Proceedings of the 2008 conference on Computing frontiers*, pages 23–32, New York, NY, USA, 2008. ACM.

[46] M. Mustafa Rafique, Ali R. Butt, and Dimitrios S. Nikolopoulos. A capabilities-aware framework for using computational accelerators in data-intensive computing. *Journal of Parallel and Distributed Computing*, 71:185–197, February 2011.

[47] M. Mustafa Rafique, Nishkam Ravi, Srihari Cadambi, Ali R. Butt, and Srimat Chakradhar. Power management for heterogeneous clusters: An experimental study. In *Proc. 2nd IEEE International Green Computing Conference (IGCC)*, Orlando, FL, July 2011.

[48] M. Mustafa Rafique, Srihari Cadambi, Kunal Rao, Ali R. Butt, and Srimat Chakradhar. Symphony: A scheduler for client-server applications on coprocessor-based heterogeneous clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, Austin, TX, USA, Sept. 2011.

[49] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[50] IBM Corp. Cell Broadband Engine Architecture (Version 1.02). 2007.

[51] Astrophysicist Replaces Supercomputer with Eight PlayStation 3s. `http://www.wired.com/techbiz/it/news/2007/10/ps3_supercomputer`.

[52] Mueller. NC State Engineer Creates First Academic Playstation 3 Computing Cluster. `http://moss.csc.ncsu.edu/~mueller/cluster/ps3/coe.html`.

[53] GraphStream, Inc. GraphStream scalable computing platform (SCP). 2006. `http://www.graphstream.com`.

[54] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven H. M. Buijssen, Matthias Grajewski, and Stefan Turek. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing.*, 33(10-11):685–699, 2007.

[55] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[56] Ashlee Vance. China Wrests Supercomputer Title From U.S. *The New York Times*, October 2010. `http://www.nytimes.com/2010/10/28/technology/28compute.html`.

[57] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD'06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM.

[58] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384.

[59] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[60] D.P.Playne K.A.Hawick, A.Leist. Mixing multi-core cpus and gpus for scientific simulation software. Technical Report CSTN-091, Computational Science Technical Note, Sep. 2009.

[61] A. Leist, D. P. Playne, and K. A. Hawick. Exploiting graphical processing units for data-parallel scientific applications. *Concurr. Comput. : Pract. Exper.*, 21(18):2400–2437, 2009.

[62] Bryan McDonnel and Niklas Elmqvist. Towards utilizing gpus in information visualization: A model and implementation of image-space operations. *IEEE Transactions on Visualization and Computer Graphics*, 15:1105–1112, 2009.

[63] Sami Hissoiny, Benoit Ozell, and Philippe Despres. A convolution-superposition dose calculation engine for gpus. *Medical Physics*, 37(3):1029–1037, 2010.

[64] Weiguo Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a gpu. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, page 8, April 2006.

[65] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD'04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM.

[66] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU'10, pages 94–103, New York, NY, USA, 2010. ACM.

[67] Yongchao Liu, Douglas L Maskell, and Bertil Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Res Notes*, 2:73, 2009.

[68] Eduard Gonzales, Alun Evans, Sergi Gonzales, Juan Abadia, and Josep Blat. Real-time visualisation and browsing of a distributed video database. In *ACE'09: Proceedings of the International Conference on Advances in Computer Enterntainment Technology*, pages 423–424, New York, NY, USA, 2009. ACM.

[69] Christian Dick, Jens Schneider, and Rüdiger Westermann. Efficient geometry compression for gpu-based decoding in realtime terrain rendering. *Comput. Graph. Forum*, 28(1):67–83, 2009.

[70] Yotam Livny, Zvi Kogan, and Jihad El-Sana. Seamless patches for gpu-based terrain rendering. *Vis. Comput.*, 25(3):197–208, 2009.

[71] J. Schneider, J. Georgii, and R. Westermann. Interactive geometry decals. In *Proceedings of Vision, Modeling, and Visualization 2008*, 2009.

[72] Alexander Kohn, Jan Klein, Florian Weiler, and Heinz-Otto Peitgen. A gpu-based fiber tracking framework using geometry shaders. volume 7261, page 72611J. SPIE, 2009.

[73] Adarsh Krishnamurthy, Rahul Khardekar, Sara McMains, Kirk Haller, and Gershon Elber. Performing efficient nurbs modeling operations on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 15:530–543, 2009.

[74] Alan Chu, Chi-Wing Fu, Andrew Hanson, and Pheng-Ann Heng. Gl4d: A gpu-based architecture for interactive 4d visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15:1587–1594, 2009.

[75] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS'03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[76] John Paul Walters, Vidyananth Balu, Suryaprakash Kompalli, and Vipin Chaudhary. Evaluating the use of gpus in liver image segmentation and hmmer database searches. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[77] Wen-mei W. Hwu, Deepthi Nandakumar, Justin Haldar, Ian C. Atkinson, Brad Sutton, Zhi-Pei Liang, and Keith R. Thulborn. Accelerating mr image reconstruction on gpus. In *ISBI'09: Proceedings of the Sixth IEEE international conference on Symposium on Biomedical Imaging*, pages 1283–1286, Piscataway, NJ, USA, 2009. IEEE Press.

[78] Maraike Schellmann, Sergei Gorlatch, Dominik Meiländer, Thomas Kösters, Klaus Schäfers, Frank Wübbeling, and Martin Burger. Parallel medical image reconstruction: From graphics processors to grids. In *PaCT'09: Proceedings of the 10th International Conference on Parallel Computing Technologies*, pages 457–473, Berlin, Heidelberg, 2009. Springer-Verlag.

[79] Xing Zhao, Jing-Jing Hu, and Peng Zhang. Gpu-based 3d cone-beam ct image reconstruction for large data volume. *Journal of Biomedical Imaging*, 2009:1–8, 2009.

[80] Po-Han Wang, Yen-Ming Chen, Chia-Lin Yang, and Yu-Jung Cheng. A predictive shutdown technique for gpu shader processors. *IEEE Comput. Archit. Lett.*, 8(1):9–12, 2009.

[81] Byeong-Gyu Nam, Jeabin Lee, Kwanho Kim, Seung Jin Lee, and Hoi-Jun Yoo. A low-power handheld gpu using logarithmic arithmetic and triple dvfs power domains. In *GH'07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 73–80, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[82] Peter Bailey, Joe Myre, Stuart D. C. Walsh, David J. Lilja, and Martin O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *ICPP'09: Proceedings of the 2009 International Conference on Parallel Processing*, pages 550–557, Washington, DC, USA, 2009. IEEE Computer Society.

[83] Hassan Shojania, Baochun Li, and Xin Wang. Nuclei: Gpu-accelerated many-core network coding. In *INFOCOM*, pages 459–467. IEEE, 2009.

[84] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *ICML'09: Proceedings of the 26th Annual International Conference on Machine Learning*, pages 873–880, New York, NY, USA, 2009. ACM.

[85] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[86] Cloud services for gpu computing, 2011. `http://www.hoopoe-cloud.com/`.

[87] Accelerator cluster, 2011. `http://www.iacat.uiuc.edu/resources/cluster/`.

[88] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI 2004*, pages 137–150, 2004.

[89] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proc. of the 17th IEEE International Conference on Parallel Architectures and Compilation Techniques*, Toronto, Canada, October 2008.

[90] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA'07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[91] Apache Software Foundation. Hadoop, May 2007. `http://hadoop.apache.org/core/`.

[92] Adam Pisoni. Skynet, Apr. 2008. `http://skynet.rubyforge.org`.

[93] Matei Zaharia, Andy Konwinski, and Anthony D. Joseph. Improving mapreduce performance in heterogeneous environments. In *Proc. 8th USENIX OSDI*, San Diego, CA, Dec. 2008.

[94] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[95] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[96] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[97] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC'02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[98] Khronos Group Std. The OpenCL Specification, Version 1.0, April 2009. Online. Available: `http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf`.

[99] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, New York, NY, USA, 2009. ACM.

[100] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *HPDC'08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200, New York, NY, USA, 2008. ACM.

[101] G. Contreras and M. Martonosi. Characterizing and improving the performance of the intel threading building blocks runtime system. In *International Symposium on Workload Characterization (IISWC 2008)*, September 2008.

[102] NVIDIA. Nvidia cuda compute unified device architecture, 2nd ed, Oct. 2008.

[103] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. pages 16–30, 2008.

[104] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPoPP'09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.

[105] Jeff A. Stuart and John D. Owens. Message passing on data-parallel architectures. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[106] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. High-performance cuda kernel execution on fpgas. In *ICS'09: Proceedings of the 23rd international conference on Supercomputing*, pages 515–516, New York, NY, USA, 2009. ACM.

[107] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[108] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA'09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 140–151, New York, NY, USA, 2009. ACM.

[109] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA'00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 161–171, New York, NY, USA, 2000. ACM.

[110] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf,

Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. *SIGARCH Comput. Archit. News*, 32(2):2, 2004.

[111] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M. S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed microarchitectural protocols in the trips prototype processor. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 480–491, Washington, DC, USA, 2006. IEEE Computer Society.

[112] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proc. 1st USENIX NSDI*, pages 365–378, San Francisco, CA, Mar. 2004.

[113] Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The Kangaroo approach to data movement on the grid. In *Proc. 10th IEEE HPDC-10*, pages 325–333, San Francisco, CA, Aug. 2001.

[114] Wael R. Elwasif, James S. Plank, and Rich Wolski. Data staging effects in wide area task farming applications. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 122–129, Washington, DC, May 2001.

[115] James S. Plank, Micah Beck, Wael R. Elwasif, Terry Moore, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: Storage in the network. In *Proc. NetStore99: The Network Storage Symposium*, Jan. 1999.

[116] M. Beck, T. Moore, J. S. Plank, and M. Swany. Logistical networking: Sharing more than the wires. In *Active Middleware Services*. S. Hariri, C. Lee and C. Raghavendra editors. Kluwer Academic, Norwell, MA, 2000.

[117] Alessandro Bassi, Micah Beck, Terry Moore, James S. Plank, Martin Swany, Rich Wolski, and Graham Fagg. The internet backplane protocol: A study in resource sharing. *Future Generation Computing Systems*, 19(4):551–561, 2003.

[118] Rich Wolski, Neil Spring, and Jim Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15(5):757–768, 1999.

[119] Houda Lamehamedi, Zujun Shentu, Boleslaw Szymanski, and Ewa Deelman. Simulation of dynamic data replication strategies in data grids. In *Proc. IPDPS*, Nice, France, Apr. 2003.

[120] Henry Monti, Ali Raza Butt, and Sudharshan S. Vazhkudai. A result-data offloading service for hpc centers. In *Proc. ACM Petascale Data Storage Workshop*, Reno, NV, Nov. 2003.

[121] Henry Monti, Ali Raza Butt, and Sudharshan S. Vazhkudai. Timely offloading of result-data in hpc centers. In *Proc. 22nd ACM International Conference on Supercomputing (ICS'08)*, Kos, Greece, Jun. 2008.

[122] Henry Monti, Ali R. Butt, and Sudharshan S. Vazhkudai. Just-in-time staging of large input data for supercomputing jobs. In *Proc. ACM Petascale Data Storage Workshop*, Austin, TX, Nov. 2008.

[123] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *Proc. 4th USENIX FAST*, pages 187–200, San Francisco, CA, Mar. 2004.

[124] Chris Gniady, Ali Raza Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *Proc. 6th USENIX OSDI*, pages 395–408, San Francisco, CA, Dec. 2004.

[125] Richard W. Carr and John L. Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. In *Proc. 8th ACM SOSP*, pages 87–95, Pacific Grove, CA, Dec. 1981.

[126] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD*, pages 297–306, May 1993.

[127] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. An optimality proof of the LRU-K page replacement algorithm. *Journal of the ACM*, 46(1):92–112, 1999.

[128] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proc. 20th International Conference on Very Large Databases*, pages 439–450, Santiago, Chile, Jan. 1994.

[129] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS*, pages 31–42, June 2002.

[130] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS*, pages 134–142, May 1990.

[131] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1360, 2001.

[132] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proc. ACM SIGMETRICS*, pages 134–143, May 1999.

[133] Nimrod Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. 2nd USENIX FAST*, pages 115–130, San Francisco, CA, Mar. 2003.

[134] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[135] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. 15th ACM SOSP*, pages 79–95, Copper Mountain, CO, Dec. 1995.

[136] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, 2001.

[137] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proc. ACM SIGMETRICS*, pages 115–126, June 1997.

[138] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. In *Proc. ACM SIGMETRICS*, pages 122–133, Atlanta, GA, May 1999.

[139] Jongmoo Choi, Sam H. Noh, Smig Lyul Min, and Yookun Cho. An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme. In *Proc. USENIX ATC*, pages 239–252, Monterey, CA, June 1999.

[140] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proc. ACM SIGMETRICS*, pages 286–295, Santa Clara, CA, June 2000.

[141] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. In *Proc. 4th USENIX OSDI*, pages 119–134, San Diego, CA, Oct. 2000.

[142] Susanne Albers and Markus Büttner. Integrated prefetching and caching in single and parallel disk systems. In *Proc. 15th ACM SPAA*, pages 24–39, Duluth, MN, June 2003.

[143] Mahesh Kallahalla and Peter J. Varman. Optimal prefetching and caching for parallel I/O sytems. In *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 219–228, Crete Island, Greece, July 2001.

[144] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A trace-driven comparison of algorithms for parallel prefetching and caching. *SIGOPS Oper. Syst. Rev.*, 30(SI):19–34, 1996.

[145] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proc. 1st USENIX OSDI*, pages 165–177, Monterey, CA, Nov. 1994.

[146] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. 2nd USENIX OSDI*, pages 3–17, Seattle, WA, 1996.

[147] Fay W. Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. 3rd USENIX OSDI*, pages 1–14, New Orleans, LA, Feb. 1999.

[148] Jim Griffioen and Randy Appleton. Performance measurements of automatic prefetching. In *Proc. Parallel and Distributed Computing Systems*, pages 165–170, Sept. 1995.

[149] Vivekanand Vellanki and Ann L. Chervenak. A cost-benefit scheme for high performance predictive prefetching. In *Proc. ACM/IEEE conference on Supercomputing*, Portland, OR, Nov. 1999.

[150] Nancy Tran and Daniel A. Reed. ARIMA time series modeling and forecasting for adaptive I/O prefetching. In *Proc. 15th International Conference on Supercomputing*, pages 473–485, Sorrento, Italy, June 2001.

[151] Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *SIAM J. Comput.*, 29(4):1051–1082, 2000.

[152] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proc. ACM SIGMOD*, pages 257–266, Washington, D.C., May 1993.

[153] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proc. USENIX Summer Technical Conference*, pages 197–207, Boston, MA, 1994.

[154] K. Korner. Intelligent caching for remote file service. In *Proc. ICDCS*, pages 220–226, Paris, France, May 1990.

[155] D. Kotz and C.S. Ellis. Practical prefetching techniques for parallel file systems. In *Proc. 1st International Conf. on Parallel and Distributed Information Systems*, pages 182–189, Miami, FL, December 1991.

[156] M.L. Palmer and S.S. Zdonik. FIDO: A cache that learns to fetch. Technical Report CS-90-15, Brown University, 1991.

[157] Chao-Tung Yang and Lung-Hsing Cheng. Implementation of a performance-based loop scheduling on heterogeneous clusters. In *ICA3PP'09: Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, pages 44–54, Berlin, Heidelberg, 2009. Springer-Verlag.

[158] Bonnie Holte Bennett, Emmett Davis, Timothy Kunau, and W. Wren. Beowulf parallel processing for dynamic load-balancing. In *Proceedings of IEEE Aerospace Conference*, pages 389–395, 2000.

[159] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous computing systems. In *Heterogeneous Computing Workshop (HCW'97)*, page 135, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[160] Eitan Frachtenberg, Dror G. Feitelson, Juan Fernandez, and Fabrizio Petrini. Parallel job scheduling under dynamic workloads. In *Proceedings of the Ninth Workshop Job Scheduling Strategies for Parallel Processing*, June 2003.

[161] Christopher A. Bohn and Gary B. Lamont. Load balancing for heterogeneous clusters of pcs. *Future Generation Computer Systems*, 18(3):389–400, 2002.

[162] Keqin Li. Optimal load distribution in nondedicated heterogeneous cluster and grid computing environments. *Journal of Systems Architecture*, 54(1-2):111–123, 2008.

[163] Bora Ucar, Cevdet Aykanat, Kamer Kaya, and Murat Ikinci. Task assignment in heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 66(1):32–46, 2006.

[164] Jorge Manuel Gomes Barbosa and Belmiro Daniel Rodrigues Moreira. Dynamic job scheduling on heterogeneous clusters. In *ISPDC'09: Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*, pages 3–10, Washington, DC, USA, 2009. IEEE Computer Society.

[165] Micah Adler, Ying Gong, and Arnold L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *SPAA'03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 1–10, New York, NY, USA, 2003. ACM.

[166] Arnaud Giersch, Yves Robert, and Frédéric Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. *J. Syst. Archit.*, 52(2):88–104, 2006.

[167] Ligang He, Stephen A. Jarvis, Daniel P. Spooner, and Graham R. Nudd. Dynamic scheduling of parallel real-time jobs by modelling spare capabilities in heterogeneous clusters. *IEEE International Conference on Cluster Computing (CLUSTER)*, 2003.

[168] Neeraj Nehra, R.B. Patel, and V.K. Bhat. A framework for distributed dynamic load balancing in heterogeneous cluster. *Journal of Computer Science*, 3(1):14–24, 2007.

[169] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC'08, pages 9:1–9:12, Piscataway, NJ, USA, 2008. IEEE Press.

[170] R. Thakur, W. Gropp, and E. Lusk. Users guide for ROMIO: a High-Performance, portable MPI-IO implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Oct. 1997.

[171] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI'10, pages 86–97, New York, NY, USA, 2010. ACM.

[172] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC'09, pages 39–48, New York, NY, USA, 2009. ACM.

[173] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. *SIGARCH Comput. Archit. News*, 39:369–380, March 2011.

[174] Tianyi David Han and Tarek S. Abdelrahman. hicuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.

[175] Dong Hyuk Woo and Hsien-Hsin S. Lee. Compass: a programmable data prefetcher using idle gpu shaders. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS'10, pages 297–310, New York, NY, USA, 2010. ACM.

[176] Nawab Ali and Mario Lauria. Improving the performance of remote i/o using asynchronous primitives. In *IEEE International Symposium on High Performance Distributed Computing*, pages 218–228, 2006.

[177] Keith Bell, Andrew Chien, and Mario Lauria. A high-performance cluster storage server. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC'02, Washington, DC, USA, 2002. IEEE Computer Society.

[178] Christina M. Patrick, SeungWoo Son, and Mahmut Kandemir. Comparative evaluation of overlap strategies with study of i/o overlap in mpi-io. *SIGOPS Oper. Syst. Rev.*, 42:43–49, October 2008.

[179] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par'96, pages 573–577, London, UK, 1996. Springer-Verlag.

[180] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'09, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[181] Lei Wang, Yong-zhong Huang, Xin Chen, and Chun-yan Zhang. Task scheduling of parallel processing in cpu-gpu collaborative environment. In *Proceedings of the 2008 International Conference on Computer Science and Information Technology*, pages 228–232, Washington, DC, USA, 2008. IEEE Computer Society.

[182] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW'99, pages 3–14, Washington, DC, USA, 1999. IEEE Computer Society.

[183] Muthucumaru Maheswaran and Howard Jay Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proceedings of the Seventh Heterogeneous Computing Workshop*, pages 57–69, Washington, DC, USA, 1998. IEEE Computer Society.

[184] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Softw. Eng.*, 3:85–93, January 1977.

[185] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4:175–187, February 1993.

[186] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13:260–274, March 2002.

[187] Benjamin Hindman, Andrew Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. Technical Report UCB/EECS-2010-87, EECS Department, University of California, Berkeley, May 2010.

[188] Chao-Tung Yang and Keng-Yi Chou. An adaptive job allocation strategy for heterogeneous multiple clusters. In *Proceedings of the 2009 Ninth IEEE International Conference on Computer and Information Technology - Volume 02*, CIT'09, pages 209–214, Washington, DC, USA, 2009. IEEE Computer Society.

[189] David Chess, Benjamin Grosof, Colim Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications*, 3:34–49, 1995.

[190] R. B. Patel and K. Garg. Pmade a platform for mobile agent distribution & execution. In *Proceedings of 5th World MultiConference on Systemics, Cybernetics and Informatics (SCI2001) and 7th International Conference on Information System Analysis and Synthesis (ISAS 2001)*, pages 287–292, Orlando, Florida, USA, July 2001.

[191] Abhishek Chandra, Micah Adler, and Prashant Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. In *Proceedings of the Seventh Real-Time Technology and Applications Symposium*, RTAS'01, Washington, DC, USA, 2001. IEEE Computer Society.

[192] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys'10, pages 265–278, New York, NY, USA, 2010. ACM.

[193] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41:59–72, March 2007.

[194] Sang-Min Park and Marty Humphrey. Predictable time-sharing for dryadlinq cluster. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC'10, pages 175–184, New York, NY, USA, 2010. ACM.

[195] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[196] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In

Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP'09, pages 261–276, New York, NY, USA, 2009. ACM.

[197] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proc. Summer USENIX*, pages 119–130, Portland, OR, June 1985.

[198] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proc. Ottawa Linux Symposium*, Ottawa, Canada, July 2003.

[199] NVIDIA Corporation. GeForce GTX 295 - A powerful dual chip graphics card for gaming and beyond., 2011. `http://www.nvidia.com/object/product_geforce_gtx_295_us.html`.

[200] A. K. Nanda, J. R. Moulic, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D'Arnora, and S. Kesavarapu. Cell/b.e. blades: building blocks for scalable, real-time, interactive, and digital media servers. *IBM J. Res. Dev.*, 51(5):573–582, 2007.

[201] Jakub Kurzak, Alfredo Buttari, Piotr Luszczek, and Jack Dongarra. The playstation 3 for high-performance scientific computing. *Computing in Science and Engineering*, 10(3):84–87, 2008.

[202] Paul Burton, Lyle Gurrin, and Peter Sly. Extending the simple linear regression model to account for correlated responses: An introduction to generalized estimating equations and multi-level mixed modelling. *Statistics in Medicine*, 17(11):1261–1291, 1998.

[203] Antoine Guisan, Thomas C. Edwards, and Trevor Hastie. Generalized linear and generalized additive models in studies of species distributions: setting the scene. *Ecological Modelling*, 157(2-3):89–100, 2002.

[204] Thrasyvoulos N. Pappas and N. S. Jayant. An adaptive clustering algorithm for image segmentation. *IEEE Transactions on Signal Processing*, 40(4):901–914, 1992.

[205] Linas Laibinis and Elena Troubitsyna. Fault tolerance in a layered architecture: A general specification pattern in b. In *SEFM'04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 346–355, Washington, DC, USA, 2004. IEEE Computer Society.

[206] Hasan Davulcu, Juliana Freire, Michael Kifer, and I. V. Ramakrishnan. A layered architecture for querying dynamic web content. In *SIGMOD'99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 1999. ACM.

[207] Isabel F. Cruz and Yuan Feng Huang. A layered architecture for the exploration of heterogeneous information using coordinated views. In *VLHCC'04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 11–18, Washington, DC, USA, 2004. IEEE Computer Society.

[208] Jon Salz, Alex Snoeren, and Hari Balakrishnan. TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-End Network Services. In *4th Usenix Symposium on Internet Technologies and Systems*, Seattle, WA, March 2003.

[209] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 11(2):215–255, 2002.

[210] Yannis Smaragdakis and Don S. Batory. Mixin-based programming in c++. In *GCSE'00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pages 163–177, London, UK, 2001. Springer-Verlag.

[211] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[212] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[213] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[214] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the Generative Programming and Component Engineering (GPCE)*, pages 125–140, 2005.

[215] Tom Mens, Pieter Van Gorp, Dániel Varró, and Gabor Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 152:143–159, 2006.

[216] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[217] NVIDIA Corporation. GeForce 9600M GT, 2008. `http://www.nvidia.com/object/product_geforce_9600m_gt_us.html`.

[218] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *IEEE Transactions on Computers*, 56(7):889–908, 2007.

[219] IBM Corp. Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification, Version 1.0, November 2005.

[220] IBM Corp. Cell Broadband Engine Software Development Kit 2.1 Programmer's Guide (Version 2.1). 2006.

[221] Intel. Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor, March 2004.

[222] Intel. Intel Atom Processor N550, Jan 2011. `http://ark.intel.com/Product.aspx?id=50154`.

[223] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: design and implementation of a power-proportional web cluster. *SIGCOMM Comput. Commun. Rev.*, 41:102–108.

[224] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC'10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 241–252, New York, NY, USA, 2010. ACM.

[225] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *IISWC'07: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, pages 171–180, Washington, DC, USA, 2007. IEEE Computer Society.

[226] Willis Lang, Jignesh M. Patel, and Srinath Shankar. Wimpy node clusters: what about non-wimpy workloads? In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN'10, pages 47–55, New York, NY, USA, 2010. ACM.

[227] David Mosberger and Tai Jin. httperf–a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26:31–37, December 1998.

[228] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press, 2008.

[229] M. Jesus Zafont, Alberto Martin, Francisco Igual, and Enrique S. Quintana-Orti. Fast development of dense linear algebra codes on graphics processors. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[230] Jens Schneider, Martin Kraus, and Rdiger Westermann. Gpu-based euclidean distance transforms and their application to volume rendering. In *Computer Vision, Imaging and Computer Graphics. Theory and Applications*, volume 68 of *Communications in Computer and Information Science*, pages 215–228. Springer Berlin Heidelberg, 2010.

[231] Jeff A. Stuart, Cheng-Kai Chen, Kwan-Liu Ma, and John D. Owens. Multi-gpu volume rendering using mapreduce. In *HPDC'10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 841–848, New York, NY, USA, 2010. ACM.

[232] Kai Buerger, Florian Ferstl, Holger Theisel, and Rüdiger Westermann. Interactive streak surface visualization on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1259–1266, 2009.

[233] Bing Bai, Jason Weston, David Grangier, Ronan Collobert, Kunihiko Sadamasa, Yanjun Qi, Olivier Chapelle, and Kilian Weinberger. Learning to rank with (a lot of) word features. *Information Retrieval*, 13:291–314, 2010.

[234] VideoLAN. x264 - A Free h264/avc Encoder, Nov 2010. `http://www.videolan.org/developers/x264.html`.

[235] Craig Kolb and Matt Pharr. Chapter 45. Options Pricing on the GPU. In *GPU Gems 2*, 2009. `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter45.html`.

[236] bing Visual Search, 2011. `http://www.bing.com/browse`.

[237] Yang Chen, Tianyu Wo, and Jianxin Li. An efficient resource management system for on-line virtual cluster provision. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD'09, pages 72–79, Washington, DC, USA, 2009. IEEE Computer Society.

[238] Christoph Fehling, Frank Leymann, and Ralph Mietzner. A framework for optimized distribution of tenants in cloud applications. *Cloud Computing, IEEE International Conference on*, 0:252–259, 2010.

[239] Afkham Azeez, Srinath Perera, Dimuthu Gamage, Ruwan Linton, Prabath Siriwardana, Dimuthu Leelaratne, Sanjiva Weerawarana, and Paul Fremantle. Multi-tenant soa middleware for cloud computing. *IEEE International Conference on Cloud Computing*, pages 458–465, 2010.

[240] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual International Symposium on Computer architecture*, ISCA'09, pages 152–163, New York, NY, USA, 2009. ACM.

[241] Michela Becchi, Surendra Byna, Srihari Cadambi, and Srimat Chakradhar. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'10, pages 82–91, New York, NY, USA, 2010. ACM.

[242] Michela Becchi, Srihari Cadambi, and Srimat Chakradhar. Enabling legacy applications on heterogeneous platforms. In *Proceedings of USENIX HotPar'10*, HotPar'10. USENIX, 2010.

[243] Victor Podlozhnyuk. Black-Scholes option pricing. In *White Paper, NVIDIA Corporation*, June 2007.

[244] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC3530: Network File System (NFS) Version 4 Protocol, 2004. `http://www.ietf.org/rfc/rfc3530.txt`.

[245] Rich Miller. Go Daddy Ad Drives Huge Traffic Spike, February 2010. `http://www.datacenterknowledge.com/archives/2010/02/08/go-daddy-ad-drives-huge-traffic-spike/`.

[246] John Treadway. The End of Over-Provisioning, June 2010. `http://cloudcomputing.sys-con.com/node/1429706`.