UNIVERSITY OF CALIFORNIA, SAN DIEGO

PMap: Unlocking the Performance Genes of HPC Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Jiahua He

Committee in charge:

Professor Jeanne Ferrante, Co-Chair
Professor Allan E. Snavely, Co-Chair
Professor Sheldon Brown
Professor Sorin Lerner
Professor J. Andrew McCammon

2011

The dissertation of Jiahua He is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____
Co-Chair

_____
Co-Chair

University of California, San Diego

2011

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First of all I would like to express my deepest gratitude to my advisor and friend, Allan Snavely. He is the one who led me into the challenging but exciting world of performance characterization and modeling with his enthusiasm and dedication. This dissertation could not have been completed without his considerate guidance, selfless support, and stimulating inspiration.

I would also like to thank my other committee members, Jeanne Ferrante, Sorin Lerner, Andrew McCammon, and Sheldon Brown, for their valuable time and insightful advices.

Most of my time in UCSD was spent with my colleagues in the Gordon team, the PMaC lab and the Grail lab. It is my honor to work with and learn from these amazing people with intellect, passion, and commitment. I really apprecate all their helps for my studies, work, and life during these years.

Lastly but not least, profound and heartfelt thanks are due to my family. No matter what happens, they are always there for me. The love and joy from my family is the most precious gift of my life.

Chapter 2, in part, is a reprint of the material as it appears in the 7th IEEE/ACM International Conference on Grid Computing (Grid'06), a joint work with Omid Khalili, Catherine Olschanowskyd, Allan Snavely, and Henri Casanova. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it will appear in the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), a joint work with Allan Snavely, Rob Van der Wijngaart, and Michael Frumkin. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10), a joint work with Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett, and Allan Snavely. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in the 2010 Teragrid Conference (TeraGrid'10), a joint work with Jeffrey Bennett and Allan

Snavely. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, is a reprint of the material as it appears in the workshop on Application of Communication Theory to Emerging Memory Technologies (ACTEMT'10) hold with Globecom'10, a joint work with Mitesh Meswani, Pietro Cicotti, and Allan Snavely. The dissertation author was the primary investigator and author of this paper.

VITA

2000	B. S. in Computer Science, University of Science and Technology of China
2003	M. S. in Computer Science, University of Science and Technology of China
2011	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

"Automatic Recognition of Performance Idioms in Scientific Applications", J. He, A. Snavely, R. Van der Wijngaart, and M. Frumkin, To appear in the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, Alaska, May 16-20, 2011.

"Predicting Disk I/O Time of HPC Applications on Flash Drives", M. Meswani, P. Cicotti, J. He, and A. Snavely, Workshop on Application of Communication Theory to Emerging Memory Technologies (ACTEMT'10) with Globecom'10, Miami, Florida, December 6-10, 2010.

"DASH: a Recipe for a Flash-based Data Intensive Supercomputer", J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely, In Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10), New Orleans, LA, November 13-19, 2010.

"Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing" (**Nominated as the best paper and the best student paper**), A. M. Caulfield, J. Coburn, T. I. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, In Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10), New Orleans, LA, November 13-19, 2010.

"DASH-IO: an empirical study of flash-based IO for HPC" (**Nominated as the best paper**), J. He, J. Bennett, and A. Snavely, In Proceedings of the 2010 Teragrid Conference (TeraGrid'10), Pittsburgh, PA, August 2-5, 2010.

"Code coverage, performance approximation and automatic recognition of idioms in scientific applications" J. He, A. Snavely, R. F. Van der Wijngaart, and M. A. Frumkin, In Proceedings of the 17th international Symposium on High Performance Distributed Computing (HPDC'08), Boston, MA, June 23-27, 2008.

"Measuring the Performance and Reliability of Production Computational Grids", O. Khalili, J. He, C. Olschanowskyd, A. Snavely, and H. Casanova, In Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (Grid'06), Barcelona, Spain, September 28-29, 2006.

ABSTRACT OF THE DISSERTATION

PMap: Unlocking the Performance Genes of HPC Applications

by

Jiahua He

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Jeanne Ferrante, Co-Chair
Professor Allan E. Snavely, Co-Chair

Performance modeling, the science of understanding and predicting application performance, is important but challenging. High Performance Computing (HPC) with large-scale applications and aggressive technologies, such as dynamic computational grids, hybrid computing platforms, and innovative storage systems, further complicates the task. This dissertation proposed and proved the hypothesis that a small number of performance primitives can be extracted from HPC applications and leveraged for fast application performance modeling and prediction even on large-scale dynamic systems. PMap: a set of methods and tools to extract, measure, and analyze performance primitives in HPC applications are proposed, implemented, and verified under these challenging environments.

Two production computational grids, Teragrid and Geon, were monitored with periodically running benchmarks for about half a year. Their performance fluctuated in the 50% range. However, simple benchmarks that serve as performance primitives can be used to predict application performance with a relative error as low as 9%.

To map program constructs to the best matched hardware components in hybrid computing platforms, an automatic idioms (performance primitives) recognition method was proposed and implemented based on the open source compiler Open64. With the NAS Parallel Benchmark (NPB) as a case study, the prototype system is about 90% accurate compared with idiom classification by a human expert. The performance of the idiom benchmarks with their corresponding instances in the NPB codes on two different platforms were compared with different methods. The approximation accuracy is up to 97%.

With the HPC data challenge and emerging storage technologies, a flash-based supercomputer DASH was designed, built, and tuned. A large parameter space was swept by fast and reliable measurements developed to investigate varying design options, and the results showed that performance can be improved by as much as $9x$ with appropriate existing technologies developed here. Finally, the PMaC framework was extended to model and predict application performance on flash storage systems. Results showed that the total I/O time can be predicted with reasonable error of 15%.

The end result of this body of work is that the performance of applications on supercomputers can be understood by mapping their performance genetics.

Chapter 1

Introduction

Performance modeling and prediction is designed to catch the behavioral characteristics of a specific algorithm-architecture or program-architecture combination. Understanding the range of characteristics is key for architecture design, program optimization, and system evaluation. As a result, performance modeling and prediction methods are widely used in almost all branches of computer science, such as architecture design [25][143][41], job scheduling [77][125], software engineering [14], compiler optimization [136], storage systems capability planning [8], and high performance computer acquisition and performance tuning [26][99].

However, it is not trivial to model and predict performance accurately and quickly even for simple sequential programs. Metrics, such as the number of floating point operations that an application contains, cannot predict actual performance very well [26]. More advanced metrics are needed as the increasing speed gap between processor and memory (and even I/O), as well as the features of modern processors like multiple instruction issuing and out-of-order execution, make memory hierarchy and instruction mix important. In parallel programs, parallelism, synchronization, contention and communication even complicate the situation further.

Table 1.1: Classification of Performance Modeling Methods

	Input		Level of detail	Analysis method
	Software	Hardware		
Analytical modeling	Pseudo code	Computational model parameters	Statement	Model analysis
Empirical analysis	Source code /Executable	Real architecture	Program construction	Profile analysis
Simulation	Executable /Trace	Specification	Instruction cycle /Trace event	Simulation /Emulation

1.1 Performance Modeling and Prediction

There are several different classification approaches for performance modeling and prediction methods in the literature [2][18][94]. This work summarizes them and classifies the methods into three categories: analytical modeling, empirical analysis and simulation, which are shown in Table 1.1.

Analytical modeling [45][135][38] is mainly used on the algorithm level and is similar to complexity analysis of sequential algorithms. There is also some work on analytical modeling of specific applications [67][80]. These models require intensive labor and expertise to construct, and are not generally applicable to most applications. An analytical model typically includes a simplified abstraction of its hardware architecture. In order to describe a specific algorithm and architecture combination, an analytical model takes pseudo code of the target algorithm and parameters of the target architecture as inputs. Then the number of algorithmic operations, such as floating-point computation and memory access, is counted on the target architecture to predict the performance. Though they are generally manual, analytical models can be used as underlying models for automatic tools such as a simulator to analyze the performance of a program [12].

Empirical analysis [26][79] usually adopts measurement, profiling, and other empirical methods to analyze and predict performance. Real programs in the form

of source code or executable and real architectures are required for measurement and analysis. For measurement, static and dynamic program analysis are used to collect program profiles, and micro benchmarks or hardware specifications are exploited to identify architecture characteristics. Different approaches will expose details on various levels of programs, such as instructions, statements and loops. For analysis, there are also many different approaches, such as convolution in PMaC model [26] and simulation in the model presented by Marin and Mellor-Crummey [79]. They are all based on program profiles and architecture characteristics collected, which is called profile analysis.

In contrast to the methods discussed above, simulation [25][36][12] usually runs the program virtually step by step driven by execution or trace. A simulation driven by execution is also called emulation, which actually executes the instructions in the program. On the other hand, a trace-base simulation takes a stored trace of a program as input and runs off of the events in the trace. The specification of the architecture is mainly embedded in the simulator with several adjustable parameters. Most processor simulators, such as SimpleScalar [25], mimic each instruction cycle of the program to determine the timing precisely. Simulators of other system components, such as I/O and communication, usually have coarser granularities performing on I/O and communication event level.

1.2 Challenges of Performance Modeling and Prediction in High Performance Computing

Performance modeling and prediction is challenging in terms of accuracy and modeling time. Simulation is often believed to be the most accurate method since it considers the details down to instruction cycles. However, its running time is usually prohibitive. Though many accelerating techniques are proposed [143][110], simulation time is still several order of magnitudes longer than that of direct execution, which usually limits the application of this method to small and medium scales.

In high performance computing (HPC), the situation is even worse. HPC

users are generally trying to solve large-scale scientific problems, which may take days, weeks, or even months to compute. Performance modeling and prediction for HPC applications is more time-consuming than the application themselves. Furthermore, to speed up these applications, the community tends to apply aggressive technologies like parallel computing, grid computing, hybrid computing, and out-of-core algorithms. The application of these technologies introduces more complexities, such as parallelism, dynamic behavior, and online resource scheduling. For instance, a dynamic computing platform like a computational grid might need on-the-fly performance prediction. Otherwise, the prediction results will become stale before they can be used. As a result, more advanced modeling and prediction techniques are required for HPC applications.

1.2.1 Parallel Computing

In parallel computing, a problem is divided in the data dimension (data-parallel) or the task dimension (task-parallel) and solved by multiple processes running in parallel on different processors to speed up the computation. To cooperate and finish the job, these separate processes usually communicate with each other to exchange data or synchronize. In shared memory systems, processes communication is through memory; while in distributed memory systems, they communicate through the interconnection network. Since shared memory communication can be modeled as an extra memory hierarchy level and distributed memory system is the mainstream nowadays, this work will focus on the latter, especially MPI [85] (Message Passing Interface) programs.

To model and predict the performance of parallel programs, people have to consider not only the details of sequential programs referred to above but also parallelism and communication. These two additional components for parallel computing systems introduce more complexities and probably longer modeling time than for serial applications. For example, to efficiently simulate parallel programs with cycle-accurate simulation, usually the simulator itself also has to run in parallel.

In the PMaC (Performance Modeling and Characterization) lab [103], we

have worked on these challenges for years and proposed the PMaC prediction framework [102] to ease and expedite the process. Figure 1.1 shows the components and the steps of the framework. Instead of using slow simulation, the PMaC framework adopts empirical analysis to achieve fast modeling and prediction. The basic idea of the frameworks is to first collect the performance characteristics of both the target architecture (called **machine profile**) and the target application (called **application signature**), and then convolve them together with an analytical model, which is expressed in a closed-form formula. To simplify the model while guaranteeing the accuracy, the framework only takes the two dominant factors, one for local performance and another for remote communication, into account.

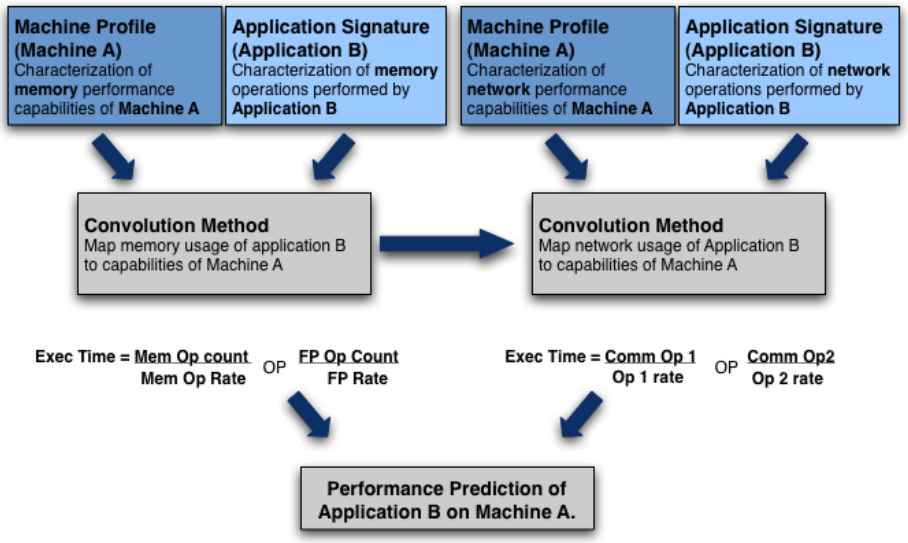

Figure 1.1: Overview of PMaC prediction framework.

To collect machine profiles, one can apply PMaC's MultiMAPS [104] benchmark and a simple MPI Ping-Pong benchmark to measure the memory and network performance of the target machine. Figure 1.2 shows some example MultiMAPS results from 3 different systems. The stair-like curves indicate the performance levels of the memory hierarchy in the modern computer architecture. To obtain application signatures, one can utilize PMaC's instrumentation tools, PMaCinst [107]/PEBIL [105] and PSiNS [106], to collect event traces and proceed coarse-grain simulations. Finally, PMaC Convolver [101] is used to combine the

machine profile and the application signature into a specific analytical model and generate the final prediction.

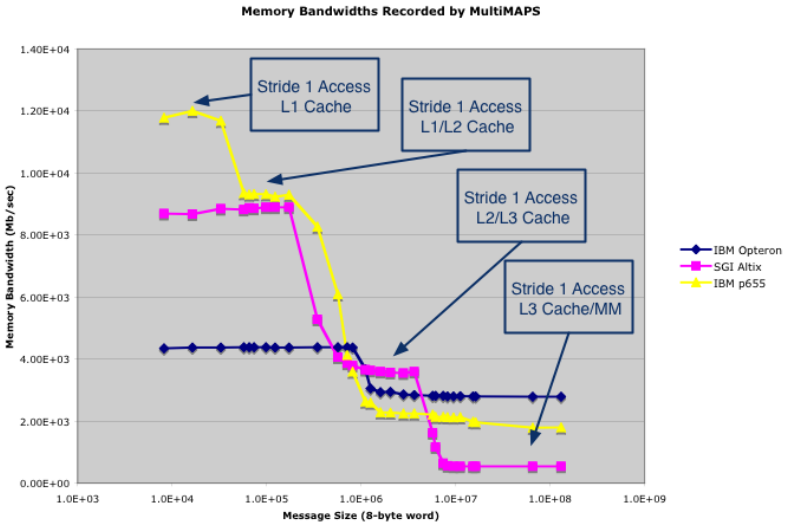

Figure 1.2: Example MultiMAPS results from 3 different System.

After years of refinement, now the PMaC framework can handle large-scale parallel programs with reasonable modeling time and accuracy [26]. However, it was not designed for the stiffer environments like grid or hybrid computing platforms. Moreover, we are in an era with emerging new storage technologies. How to quickly predict application performance across different storage technologies is also a challenge. The following sections will discuss these topics in details.

1.2.2 Grid Computing

Computational grids are dynamic in nature. First, grid resources are usually geographically distributed. As a result, they are connected together through long distance networks, even public wide area networks. The available bandwidth for a specific application is not stable. Sometimes, users may even experience failures. Second, grid resources can be shared by multiple users. The login nodes for compilation and data staging are usually shared. Even compute nodes, especially some "fat" nodes with a number of processors and large memory, can be shared.

Application performance running on shared resources will be affected by other workloads. Finally, grid resources are heterogeneous, under different administration domains, and probably in different time zones. These factors make the loads of different resources un-balanced. The performance of an application running on the same resource at different time, or on different resources at the same time can vary dramatically. Sometimes, a resource can be completely un-available because of system maintenances or hardware/software failures.

In this work, two production grid platforms, TeraGrid [132] and Geon [116], were monitored and benchmarked for about half a year. The results showed that these production grids are rather unavailable, with success rates for benchmark and application runs between 55% and 80%. It is found that performance fluctuation was in the 50% range. There is no one performance or reliability of a grid; there is only a continually evolving time-series of performances and reliabilities that may be observed and recorded. For this kind of dynamic environments, appropriate modeling methods like online prediction are needed.

Furthermore, middlewares are widely applied for computational grids and further complicate the situation. Computational grids subsume traditional compute, storage, and data acquisition resources by federating them. These platforms hold the promises of increased capacity and performance, which users/applications should achieve by the use of high-level software abstractions. Furthermore, the goal of the middleware infrastructure is to make it possible for users not to be concerned about specific resource details, but rather to operate at a higher level and perform tasks such as "find sources of data matching this description, find some suitable compute platforms to carry out a specified computation on the data, store the result in a suitable data archive and return a virtual handle to it". However, under the covers many resources and middleware services are involved and although the high-level interface may be convenient it is difficult to understand the performance and availability characteristics of such systems without appropriate benchmarking and modeling.

1.2.3 Hybrid Computing

Hybrid hardware is emerging with different components designed to speed up different program constructs. RoadRunner [15] is the first peta-flops supercomputer with hybrid hardware: Opteron ® cores plus Cell ® processors. GPU computing is becoming more and more important. The first place of the current Top500 list [133] is GPU machine, Tianhe-1A. How to recognize suitable program constructs for GPU execution automatically will be critical for the usability and efficiency of these GPU machines. FPGA machines will have even more diversities. One example is Convey ® HC-1 [22], which provides a handful application-specific co-processors called personalities for Intel ® processors. To explore the potential of this kind of hardware requires automatic recognition of different program constructs (personalities) and mapping them to corresponding hardware components. In one of our un-published works, we were able to recognize the Gather/Scatter constructs in real applications and send it to the Convey FPGA accelerator thus speeding as much as 20x.

On the road to exa-scale supercomputing, performance is not the only concern. In fact, power consumption is becoming the limit for scalability. To solve the problem, our colleague Professor Andrew Chien has proposed a new paradigm called 10x10 [33]. The basic idea is to integrate 10 different specialized cores onto one chip and use the best matched ones for specific program constructs. How to recognize these program constructs and map them to corresponding cores is a real challenge.

The key issue here is how to model and predict the performance of a specific program construct on different hardware components. It will be ideal to build up a connection between the performance and the structure of program constructs. In this way, one can easily identify appropriate program constructs by their structure and schedule them to proper hardware components.

1.2.4 New Storage Technologies

HPC applications are becoming more and more data-intensive. This work has participated in and analyzed some user interviews [123][122][70][121] and found

two important data-intensive applications categories: data mining and predictive science [129][48][11]. Both of these two kinds of applications share the same characteristics: they are dominated by small random data access (especially read) patterns.

To achieve satisfying performance for these kinds of applications, short latency is the key. Unfortunately, the current architecture was not designed for these kinds of latency-critical applications. Figure 1.3 shows the average latencies of the existing memory storage hierarchy. As may be noticed, there is a 5-order-of-magnitude latency gap between memory and spinning disks.

To fill the latency gap, one can make use of remote DRAM memory across the network interface, which is 3-order-of-magnitude faster than spinning disks. However, it requires special hardware [128] or software [114][1][29] to support. Adopting new storage technologies like lash-based SSD [3][31][28][50][98][84] is another choice, which is 2-order-of-magnitude faster than spinning disks and is promising to take the place of them.

More new storage technologies, such as PCM and STTM [27], are coming.

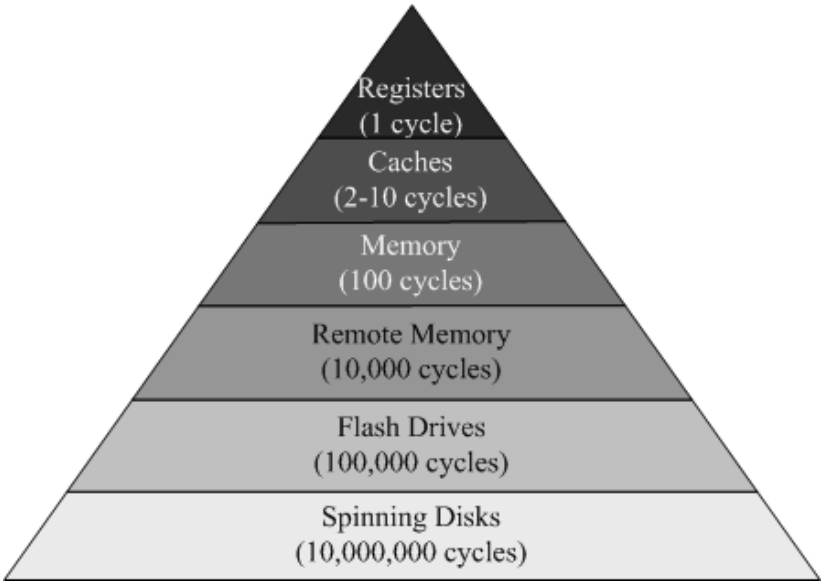

Figure 1.3: The existing memory storage hierarchy. There is a 5-order-of-magnitude latency gap between memory and spinning disks. One solution is to adopt distributed shared memory and flash drives to fill the gap.

However, transferring to a new technology can be expensive and risky. People would like to understand what the performance gain will be before they investigate. How to model and predict application performance on these new storage technologies is an interesting research topic further expounded below.

1.3 PMap overview

In scientific applications, space, time, local and long-range interactions are represented by abstract programming constructs. Despite many different implementations, there are fundamental similarities in these abstractions: space is usually modeled by structured or unstructured grids, time is often modeled by the outermost iteration loop (time step), and interactions are modeled by systems of equations that are solved by iterative explicit or implicit methods. As a result, a number of similar programming constructs can be found in various scientific applications [46].

In grid computing, resources are heterogeneous and designed for specific use scenarios. Large and complex grid applications can make use of the specialties of these resources and schedule various components to matched resources. For example, an application may acquire data from some sensors or a telescope, transfer the data to a peta-scale supercomputer for simulation or data processing, then visualize the output on a visualization cluster, and finally store the results into an archive system. As a result, a number of similar work flow structures can be identified in various grid applications.

In data-intensive applications, often out-of-core algorithms are adopted and I/O performance will dominate. Basically, I/O performance is decided by type (read/write), pattern (sequential/random), request size, and queue depth [7]. I/O streams with these four similar characteristics result in similar bandwidth and IOPS (I/O operations Per Second). As a result, a number of similar I/O streams can be seen in different data-intensive application.

All of these programming constructs, work flow structures, and I/O streams share the same feature: they are primitive components of applications and repre-

sent the critical structure related to the application performance. In this dissertation, these programming constructs, work flow structures, and I/O streams are called **performance primitives**. A fundamental hypothesis of this dissertation is that the application performance can be correlated to the performance of these primitives and this work has developed a set of methods and tools to extract, measure, and analyze these performance primitives. This set of methods and tools is called **PMap**. In the rest of this section, how PMap can be used to solve the challenges will be outlined.

1.3.1 Online Performance Prediction for Computational Grids

Chapter 2 introduces the work on performance characterization and prediction for computational grids [68]. To understand the performance fluctuation and reliability of production computational grids, the test harness and reporting framework Inca [58] was applied for deploying probe and application benchmarks on two production computational grids: TeraGrid [132] and Geon [116]. The probe benchmarks were designed to capture the performance of every stage of typical grid application work flows. Both probe and applications benchmarks were executed periodically for about half a year and the performance series were collected. With these data, one can characterize the performance behaviors of various grid work flow stages and find out the dominant fluctuation factor. Furthermore, because of the internal structure similarity between the probe and the application benchmarks, it is possible to make the probe application the performance primitive and develop an online performance predictor for the application benchmark. By this means, though computational grids are dynamic and vary a lot, one can still predict the application performance with quick probe runs.

1.3.2 Performance Idioms Recognition for Hybrid Computing Platforms

Chapter 3 discusses the work on performance idioms recognition [54]. Extraction, measurement and analysis of representative program constructs can result in a good understanding of the application performance and suggesting how best to map the applications to computer architectures. The performance analysis community has long realized this and utilized the common programming constructs and typical application kernels as benchmarks. But how to quantify the representativeness of benchmarks remains a hard problem. Usually the selection of benchmarks is to some extent subjective and decided by human sense. This work tries to identify automatically these common constructs or data flow patterns, such as stream, transpose, reduction, random access and stencil, in real applications, along with their coverage coefficients. To this end, an automatic analysis tool based on the open source compiler Open64 [96] was developed. Since these constructs are extracted with program internal structure and coverage coefficients in mind, they can work as the performance primitives of real applications. This allow us to derive application requirements for new hardware based on the analysis of a few simple primitives, as well as to map program constructs to appropriate hardware components in hybrid computing platforms.

1.3.3 Performance Characterization and Modeling of Flash Storage Systems

In chapter 4, 5, and 6, work on flash storage systems [53][52][83] are presented. HPC applications are becoming more data-intensive. At the same time, new storage technologies are emerging. To leverage these storage technologies to win the data challenge, we designed, built, and tuned a flash-based supercomputer called DASH. Since the existing hardware and software were designed without flash drives in mind, special system designs and technical workarounds had to be adopted to accommodate these new drives. To further understand the performance behaviors of flash drives, a systematic investigation with 16,800 tests was applied

for the flash storage system. With this detailed investigation, a complete picture of how flash drives interact with other system components was obtained. Finally, the PMaC framework was extended to model and predict application performance on flash storage systems. As performance primitives, different I/O streams were extracted from the target applications. Then the application performance can be mapped to the performance of these primitives with specific analytical model.

Chapter 2

Performance and Reliability of Production Computational Grids

Computational grids are dynamic in nature. Composing grid resources, such as network connections, compute nodes, storage systems, are usually heterogeneous, geographically distributed and shared by multiple users. As a result, their performance can vary dramatically. Moreover, middlewares are widely applied for computational grids. These middlewares ease the usage of the complicated systems. However, they hide the details of the systems and make performance evaluation and modeling more difficult.

This work overcame these dynamism and middleware issues with four contributions: (i) presented a generic grid measurement infrastructure, which was deployed on two state-of-the-art grids for several months; (ii) quantified the availability of the hardware and the middleware infrastructure in both platforms; (iii) quantified the magnitude and the sources of performance fluctuations in both platforms; and (iv) found that the performance experienced by simple "benchmark probes" can be used to predict the performance of a typical application with relative error as low as 9%.

Inca [58], a test harness and reporting framework, was used for periodically capturing and recording time-series of grid performance and availability metrics. Using Inca, the GrASP (Grid Assessment Probes) [34] benchmark probes and the PreCo (from [88]) application were deployed on two state-of-the-art grid plat-

forms: TeraGrid [132] and Geon [116]. Sections 2.1, 2.2 and 2.3 describe the measurement infrastructure, the target platforms, and the experimental methodology. Section 2.4 presents detailed performance and reliability data for both grids. Section 2.5 presents and evaluates several application performance prediction methods. Section 2.6 discusses related work and Section 2.7 concludes the chapter with a brief summary of results.

2.1 Measurement Infrastructure

It was found that, in order to collect the type of data needed to characterize the performance and reliability of a computational grid in a meaningful way, three components are required: (i) a set of benchmark probes that exercise basic grid functionality and that collect timing information and error messages as they run; (ii) an actual Grid application for comparison to the probes; and (iii) a framework for periodically running the probes and application, for archiving results, and for providing a way to query the results. These requirements were met by the GrASP probes, PreCo, and Inca respectively.

2.1.1 Grid Assessment Probes

The Grid Assessment Probes (GrASP) [34] are designed to serve as simple grid application kernel exemplars as well as a set of diagnostic tools. They test and measure performance of basic grid functions including file transfers, remote execution, and Grid Information Services response. All probes perform the same set of initial operations (check for a valid grid proxy, authenticate to all involved resources, check for disk space availability, etc.). This chapter adopts the two probes described below

Circle Probe – The Circle probe takes a 100 MB file and passes it in a ring around a given set of grid nodes, performing a checksum at each step along the way to ensure that the file has come across intact. As a final step, the file is transferred back to the originator, and a simple diff is applied to validate that the

file is identical to the original. There can be any number of nodes involved in the probe. This probe is meant to emulate an application performing a token-passing operation around grid sites.

Gather Probe – The Gather probe transfers 100 MB data files in parallel from any number of source nodes to a single compute node. A computation is performed on the input files and a single 500MB output file is generated and transferred to a single destination site. This probe is meant to emulate an application that performs a data aggregation operation across grid sites.

2.1.2 PreCo

PreCo (also called Transform-based Back Projection (TxBR)) is a research code used by the National Center for Microscopy and Imaging Research [88]. The computation component of this code uses a back projection algorithm to take 2-D images collected from an electron microscope to generate 3-D images. PreCo is a good exemplar of grid applications because data acquisition is potentially geographically separated from computation, computation is essentially embarrassingly parallel (thus able to efficiently utilize distributed parallel computing resources), while visualization from the results of computation is again potentially geographically separated from the previous two steps.

2.1.3 Inca

Inca [58] is a flexible framework for the automated testing, benchmarking and monitoring of Grid systems. Originally developed for use within the TeraGrid [132] project Inca has been generalized and is in use on other computational grids including Geon [116] and DEISA [40], it is included with the NMI [91] R7 release. Inca includes mechanisms to schedule the execution of information gathering scripts, and to collect, archive, publish, and display data. Inca supports a diverse set of use cases including, service reliability verification, monitoring, benchmarking, site interoperability certification, and software stack validation.

For gathering data Inca makes use of a Reporter functionality further described in [59]. A Reporter interacts directly with a resource to perform a test, benchmark, or query. For example, a Reporter can publish the version of a software package or perform a unit test to evaluate software functionality. An alpha version of Inca 2.0 prior to its official release was adopted and the configuration included a single Reporter Manager running on tg-login.sdsc.teragrid.org, three Reporters (which were implemented for the two GrASP probes and for PreCo), and the Depot which is the Inca measurement database.

2.2 Testbeds

TeraGrid – TeraGrid [132] aggregates resources at eight partner sites to create an integrated, persistent computational grid. Deployment of TeraGrid was completed in September 2004, bringing over 40 teraflops of computing power and nearly 2 petabytes of rotating storage into production, interconnected at 10-30 gigabits/second via a dedicated national network. All resources run the TeraGrid Common Software Stack (CTSS) which includes the Globus Toolkit version 2 (GT2). The resources in the grid include mostly clusters of Itanium2 IA-64, Alpha EV68, Itanium IA-32, IBM Power3-II.

Geon – The GEON grid [116], targeted to Earth Sciences applications, aggregates resources over fifteen institutions. Each institution runs a "GEON-grid" host, which provides an entry point into the system, runs a reference GEON software stack which includes the Globus Toolkit version 3 (GT3), and may be a gateway to local Data nodes and/or Compute nodes. The hardware consists of commodity X86 systems purchased from Dell and ProMicro with various racks and switches from Dell.

2.3 Experiments

Data was collected on both TeraGrid and Geon resources between September 6, 2005 and March 20, 2006, providing us with approximately 6 months of data on the TeraGrid and 3 months of data on GEON.

2.3.1 Inca Configuration

Table 2.1: Resources used in GrASP and PreCo Configurations on TeraGrid.

Hostname	Physical Location
tg-grid1.uc.teragrid.org	ANL, Chicago, IL
tg-login1.iu.teragrid.org	IU, Bloomington, IN
tg-login.ornl.teragrid.org	ORNL, Oak Ridge, TN
lonestar.tacc.utexas.edu	TACC, Austin, TX
tg-login.purdue.teragrid.org	Purdue, West Lafayette, IN
tg-login.ncsa.teragrid.org	NCSA, Urbana-Champaign, IL
tg-login.sdsc.teragrid.org	SDSC, San Diego, CA

TeraGrid Resources

The Gather probe was run over the seven TeraGrid Resources shown in Table 2.1. Data source nodes were ANL, IU, ORNL, TACC, and Purdue, the compute node was NCSA and the destination node was SDSC. The NCSA compute node has a cluster of Intel ® Itanium2 ® 1.3 GHz processors. A 100 MB file was generated and transfered from each of the source nodes to NCSA where a computation was executed and a file approximately 500 MB in size was then transfered to SDSC. The Circle probe included the same seven nodes, each transferring a 100 MB file to the next. The order of transfers was IU, ORNL, Purdue, TACC, SDSC, NCSA, ANL and then back to IU.

The PreCo application was run on TeraGrid across a smaller set of resources including Purdue, NCSA and SDSC. Data was sent from Purdue to NCSA where a computation took place and the result was then sent to SDSC for storage.

Geon Resources

Table 2.2: Resources used in GrASP and PreCo Configurations on Geon.

Hostname	Physical Location
utepgeon01.utep.edu	UTEP, El Paso, TX
agassiz.la.asu.edu	ASU, Phoenix, AZ
geon06.sdsc.edu	SDSC, San Diego, CA
geonnet1.mines.uidaho.edu	UI, Moscow, ID
geongrid.rice.edu	RICE, Houston, TX
geongrid.geo.arizona.edu	Arizona, Tucson, AZ
cgrid0.geol.iastate.edu	ISU, Ames, Iowa

The GrASP probes were ported to use GT3. The Web Services (WS) GRAM server is used to submit jobs to remote resources and GridFTP is used to transfer files (as on the TeraGrid). At the time of benchmarking, the PBS scheduler on the compute resource was not configured; instead all jobs were forked and ran on the login node. PreCo requires its compute processes to be submitted to a batch scheduler, and thus it was not possible to schedule PreCo on Geon.

Both the Gather and Circle probes were run on seven Geon resources. For the Gather probe, UTEP, ASU, SDSC, UIDAHO and RICE acted as data sources. The compute node is Arizona and the results node is CGRID. The login node of the compute node, Arizona, has two hyperthreaded Intel ® Xeon ® 2.80GHz processors. The same source file and results file sizes are used as on the TeraGrid. The Circle probe has the same transfer file size as on TeraGrid and has the following transfer order: ASU, SDSC, UIDAHO, RICE, Arizona, CGRID, UTEP and finally back to ASU.

2.3.2 Schedule Configuration

Initially the probes were scheduled to run hourly one after the other. The application was run simultaneously with the probes, immediately following the probes and thirty minutes after the probes in order to find any instantaneous, short delay, or long delay correlations between runtimes. This schedule was repeated

continuously throughout the data gathering period. After approximately three months of measurements were taken with this schedule, it was changed to run Gather and PreCo simultaneously every half hour in order to collect more data for evaluating application performance prediction methods.

2.4 Performance and Availability

2.4.1 TeraGrid

Performance

Measuring the performance of the GrASP probes on TeraGrid involved measuring the time spent transferring 500 MB of data to a compute site (100 MB from five separate sites in parallel), time spent in batch scheduler queues, time spent computing and time spent transferring a 500 MB result file.

Tables 2.3 and 2.4 show statistics for each step of the Gather probe over six months. Table 2.3 shows statistics for GrASP-specific steps, while Table 2.4 shows statistics for those steps that any application would require. The GrASP-specific steps are ones that a real grid application would probably not require on every run (e.g., data generation and staging, compilation). Trends were similar for the Circle probe and are not shown here. We can see in the tables that the two leading causes of variability (i.e., the highest standard deviations relative to the means) are: (i) initialization and finalization operations on the "login" node (initialization, staging data, building executable, and cleanup); and (ii) queue wait times. This was expected as the number of users actively working on a login node varies greatly over time and queue wait times are known to exhibit variable and non-stationary behaviors [118].

Table 2.4 shows three separate totals: (i) total probe runtime; (ii) probe runtime excluding setup/cleanup; and (iii) probe runtime excluding setup/cleanup and queue wait times. Total probe runtime has an average of 227.62 seconds and a standard deviation of 116.02 seconds, which is approximately 50% of the average runtime, a large standard deviation. For illustration purposes, Figure 2.1 shows

Table 2.3: Statistics on the Execution of GrASP Gather on TeraGrid: GrASP-Specific Steps Only (in Seconds).

Statistic	Init.	Stage Data	Build Executable	Cleanup
Average	36.03	38.41	15.35	10.76
Min	26.11	33.48	4.44	3.14
Max	847.85	418.47	70.46	413.46
Stdev	37.14	21.18	3.91	20.37

Table 2.4: Statistics on the Execution of GrASP Gather on TeraGrid: Steps Relevant to a Grid Application Only and Totals (in Seconds).

Statistics	Average	Min	Max	Stdev
Transfer Data to NCSA	8.38	5.28	194.09	9.20
Compute	18.40	17.99	61.21	2.04
Queue Wait Time	84.53	25.94	1008.60	101.13
Transfer Results	15.72	13.38	130.53	7.76
Total Runtime (i)	227.62	145.37	1142.66	116.02
Runtime w/o Setup/Cleanup (ii)	127.05	63.51	1055.52	102.46
Runtime w/o Setup/Cleanup/Q (iii)	42.51	36.87	260.31	14.36

the frequency of the probe's runtime falling between x standard deviations of the mean. This histogram is skewed to the right; runtimes are clustered near the mean but there is a significant probability of the occasional runtime exceeding the mean by several standard deviations. After removing the setup/cleanup steps, which are specific to the GrASP probes, the total average runtime is 127.05 seconds with a standard deviation of 102.46 seconds. This is 80% of the runtime, an even larger standard deviation due to the fact that queue time is then a larger percentage of the total runtime. As seen in Table 2.4, when the GrASP-specific stages and the queue waiting times are removed, the total execution time is 42.51 seconds with a standard deviation of only 14.36 seconds or 33% of the mean. This demonstrates that the middleware and network infrastructure of the TeraGrid perform rather consistently; a major factor in observed performance variability comes from queue wait times.

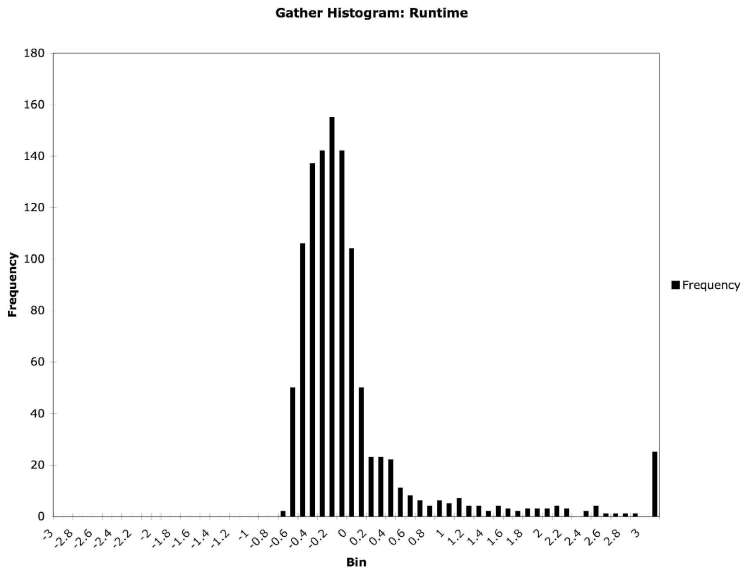

Figure 2.1: Frequency of Gather probe runtime on TeraGrid falling between x standard deviations of the mean.

Failures

An environment like TeraGrid presents many opportunities for a distributed application to fail. The GrASP probes were configured to run over seven distinct resources in the grid and a successful run required all seven resources to be up and the network connecting them to be operational. All error messages related to scheduled preventive maintenance have been filtered out. The news.teragrid.org web site was used to obtain the dates to be excluded from the measurements.

Over the first 6 weeks that the GrASP probes where running on TeraGrid resources, a success rate of approximately 58% on the gather probe and 78% on the Circle probe were observed. Figure 2.2 shows the types of errors and their counts. (The error descriptions are cryptic, please see Appendix A for detailed error descriptions). Two errors occurred far more often than the rest. The first is a proxy error, which occurred over a period of a few days when the proxy necessary for running the tests was not renewed. This error is considered a user error and is filtered out when computing overall success rate. The second is a known GT2 error that occurs frequently. The error has been fixed in later releases of Globus (however there are no plans to patch the problem in the version currently in use

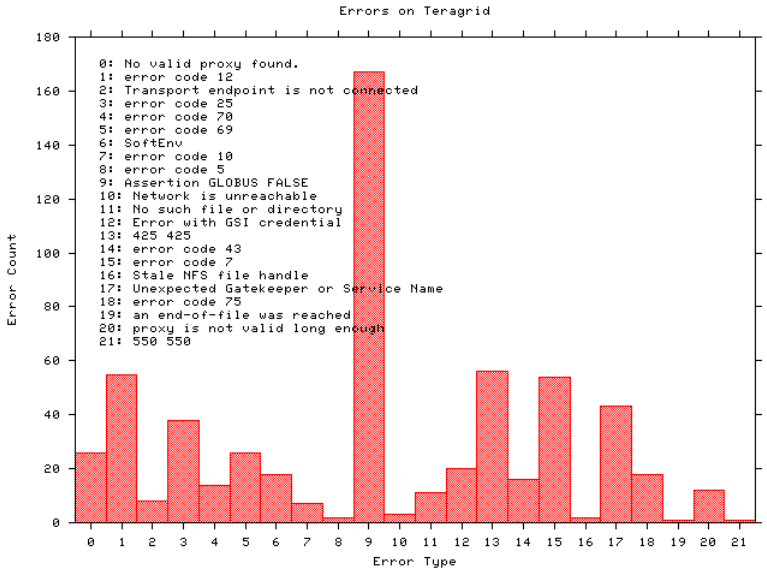

Figure 2.2: Number of Gather and Circle Errors on TeraGrid.

on TeraGrid).

The middleware errors tend to surface for a block of time and after being corrected do not reappear again. Most errors follow this pattern. Only the known GT2 error, softenv error and the error with code 12 occur periodically.

2.4.2 Geon

Results on the Geon grid differed qualitatively and quantitatively from those on TeraGrid. The heterogeneity of the TeraGrid presents great challenges to stability and software capability. The Geon grid benefits from dedicated hardware and a homogeneous software stack. However, our deployment uses GT3, which is a recently implemented web services based implementation of GT. The errors and the variability in performance due to this new middleware infrastructure are apparent in the measurements. Furthermore, while TeraGrid has dedicated network links between most of their sites, Geon uses a shared network. Therefore, one can see more variability in the transfer times on Geon. Finally, a major difference between the two grids is that Geon does not use batch schedulers but instead time slices compute resources in an interactive fashion.

Table 2.5: Statistics on the Execution of GrASP Gather on Geon: Steps Relevant to a Grid Application Only and Totals (in Seconds).

Statistic	Transfer Data to UA	Compute	Job Startup	Transfer Results	Total Runtime	Runtime w/o Setup/ Cleanup
Average	29.12	10.58	38.40	15.88	369.01	93.95
Min	21.02	10.15	29.85	14.31	324.80	84.66
Max	81.30	22.67	69.58	189.03	720.24	286.01
STDEV	11.68	0.94	2.14	9.00	34.79	15.63

Performance

Table 2.6: Statistics on the Execution of GrASP Gather on Geon: GrASP-Specific Steps Only (in Seconds).

Statistic	Init.	Stage Data	Build Executable	Cleanup
Average	96.66	70.26	33.28	74.84
Min	73.30	63.14	29.54	50.14
Max	181.38	139.87	62.04	427.74
STDEV	9.30	8.39	7.50	20.21

Tables 2.5 and 2.6 show statistics for the execution of Gather on Geon as for the TeraGrid, minus the time to start the job. For illustration purposes, Figure 2.3 shows the frequencies of individual execution times for the Gather probe on Geon that fall within x standard deviations of the mean. The statistics displayed in Table 2.6 demonstrate that the time needed for steps on the login node (initialization, staging, building the executable and cleanup) are fairly consistent. Compared to the results on the TeraGrid, execution times are longer on Geon but have a lower standard deviation. For example, the initialization step on the TeraGrid, has a average runtime of 36.03 seconds with a standard deviation of 37.14 seconds, whereas on Geon the average runtime is 96.66 seconds with a standard deviation of 9.30 seconds (less than 10%). The runtime on the TeraGrid login

nodes for these steps vary so much because they can get heavily loaded, causing occasional longer runtimes, but the average runtime is faster because GT2 is used for job submission. On Geon, the runtime on the login nodes does not vary much because there are only a few users on this new grid; nevertheless the runtime is longer because of the added overhead of using the Web Services GRAM server to start a remote job. Figure 2.3 and Table 2.5 show that a large number of executions fall outside of a standard deviation of the average runtime. This variability is due to the fact that Geon does not have a dedicated network and the runtime of Gather depends on the current network traffic.

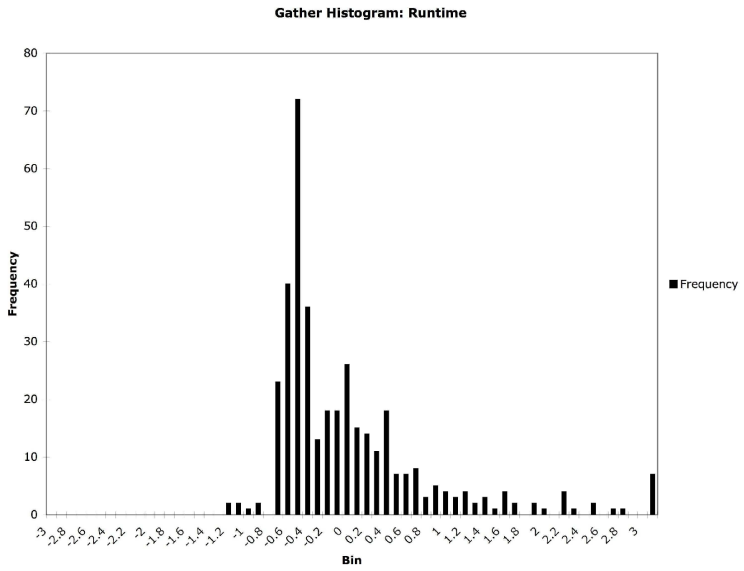

Figure 2.3: Frequency of Gather probe runtime on GEON falling between x standard deviations of the mean.

Failures

It is found that reliability on the Geon grid was much more dependent on the reliability of the middleware than that of the hardware. As mentioned previously, GT3 was used, which is essentially a web services implementation of GT2. A smaller number of error messages were returned from this version of GT. Figure 2.4 shows the error messages that occurred. The vast majority of errors were difficult to track (either just timing out or not returning any error message).

Careful sleuth work uncovered many of these errors are caused by a problem with GT3 cleaning up processes after a failure: once a failure has occurred and GT3 runs out of processes, the software just waits for a connection that cannot be created.

The Gather and Circle probes experienced a combined success rate of 56% on Geon. In most cases once an error with GT3 occurred it took human intervention to correct it and therefore the same error would happen hourly for a long period of time. It was found that most errors happened continuously for a few days before being corrected. (Appendix B lists the full error messages.) These errors did not occur frequently. As can be seen in the histogram most of them only occur a few times. The errors that occurred many times over large periods of time generally did not generate an error message and they either died silently or the Inca framework timed them out.

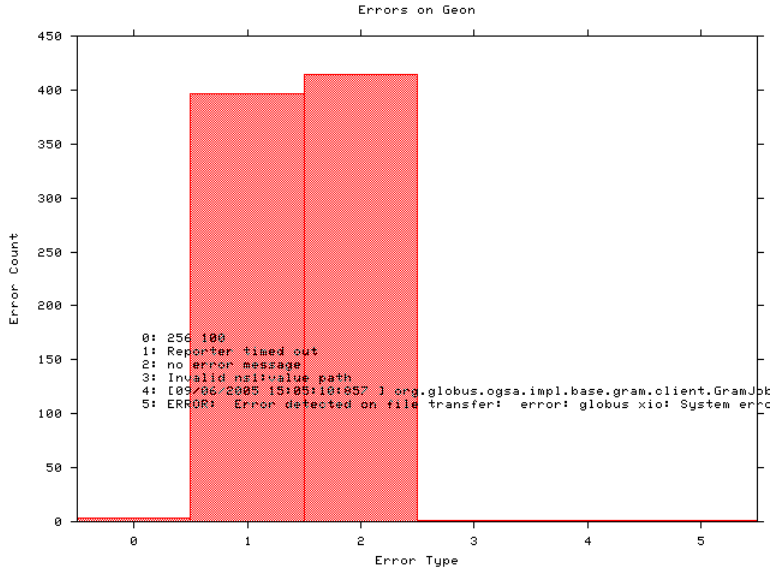

Figure 2.4: Number of Gather Errors on Geon.

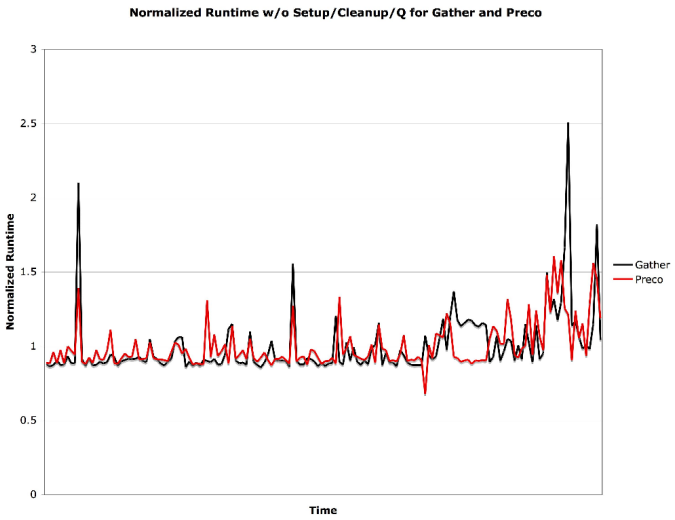

Figure 2.5: Normalized Gather and PreCo measurements (excluding the setup, cleanup and queue wait times) vs. time.

2.5 Prediction of Application Performance based on Benchmark Measurements

The schedule of probe and application benchmark execution included times when the Gather probe and PreCo application were started simultaneously, allowing us to compute the correlation of their performance. This correlation, ignoring the GrASP specific steps, to be 0.32. This correlation is low because the queue wait time, which has been shown to be a significant source of variability, accounts for most of the overall runtime. The runtimes without the GrASP specific steps and the queue wait time have a correlation of 0.54. Although all these correlation coefficients are fairly low, the trends can be seen more clearly in Figure 2.5, which shows the normalized runtimes of Gather and PreCo counting just the time to transfer the data file, run the computation and transfer the results file. This figure provides some evidence that a slow Gather runtime almost always corresponds to a slow PreCo runtime. In what follows this work investigate whether probe measurements can be used to predict an application's performance.

2.5.1 Prediction Methods

To predict the performance of an application, it will be great to predict the next point in a time series of application runtimes. However, generating the time series requires the application to run periodically, which is resource-consuming and impractical, especially for large-scale applications. (Not many people would want to regularly run their application just to be able to predict the next runtime.) To solve the problem, this work try running the light-weight benchmark probes instead and propose two kinds of methods to predict the performance of applications from time series of the probe runtimes. Figure 2.6 provides a taxonomy starting at the root of the tree with 3 classes of methods: 1) time series prediction (TSP) which attempts to predict the next application runtime directly from a time series of previous application runtimes, 2) linear regression prediction (LRP) and 3) hybrid (TSP + LRP) methods. The latter two methods attempt to predict the application runtime from time series of probe runtimes and can exploit variable or fixed training set to train their prediction functions. Variable training sets can have different sizes. This work only considers two training set sizes: the full history or the last 10 measurements. Methods with fixed training sets use the first x measurements to predict all following measurements. Details of all these methods are listed below.

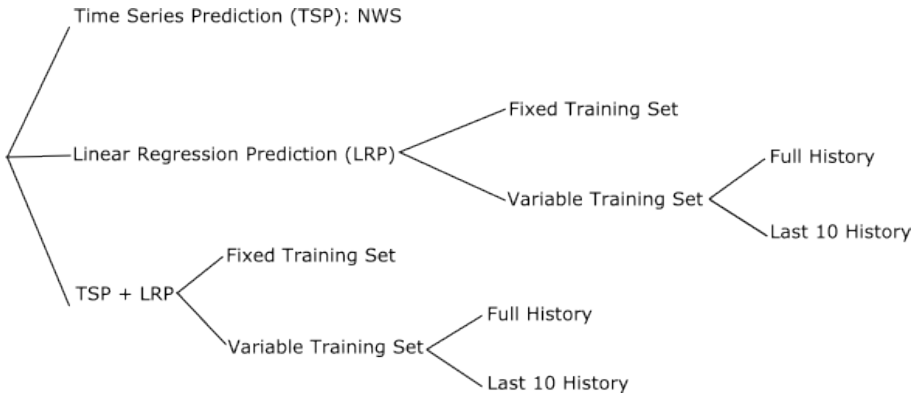

Figure 2.6: Taxonomy of Prediction Methods

- **Time Series Prediction (TSP)** In this method, the timings of the PreCo application are regarded as a time series and the NWS forecast tool is used to

predict the performance of future runs. This uses no knowledge of the Gather probe measurements; predictions are based only based on the previous PreCo runtime history.

- **Linear Regression Prediction (LRP)** In this class of methods, it is assumed that the relation between the performances of Gather and PreCo is a linear function and one can manage to calculate the coefficients. Then Gather is run periodically to collect a time series and use the last measurement to first predict the next Gather runtime and then calculate the predicted runtime of PreCo. Fixed or variable length training sets (portions of the previous history time series) can be used to create the linear regression function.

  - **Variable Training Set** With this method, each prediction has its own specific training set. These sets can have different sizes. This work only considers two situations: the full history or the last 10 measurements. When using the **full history** to make a prediction, all previous measurements of both Gather and PreCo from 1 to $n-1$ are used to predict the nth PreCo measurement. For the method using the **last 10 measurements**, only the last 10 measurements of Gather and PreCo are used to calculate the coefficients of the linear function. Either way, this method requires PreCo to run periodically with Gather; this is impractical in most real application scenarios but studied here to reveal the predictable relationship between the performances of probes and applications.

  - **Fixed Training Set** Using the fixed-training-set method, first the linear function is created using the first x measurements of both Gather and PreCo. Unlike the variable-training-set method, the linear function is not recreated for each new prediction; the linear function is used to predict all future PreCo measurements. The advantage of this method is that it is not necessary to regularly run the application; one only need to run the application when producing the training set, to calculate out the coefficients of the linear function; subsequently only the probe is

executed periodically and use that time series to make future runtime predictions for the application.

- **TSP + LRP** The above LRP methods can only be used to analyze the predictability of the application performance offline since it requires the measurement of the probe to "predict" the performance of the application submitted at the same time. To create a practical predictor scheme, the TSP and the LRP methods are combined. First, one can apply the TSP method to the time series of the probe, Gather, to predict its performance at the future time. Then the LRP method is used to predict the performance of the application, PreCo, with the previous prediction as input. The training set classification is the same as above. The hybrid method using fixed training set is a practical online prediction method without any requirement for the application to run periodically.

2.5.2 Prediction Results

For each method, its average relative error was computed (i.e., absolute error divided by actual observation) when predicting all of the observed application execution times. Table 2.7 shows these relative error and coefficient of variance (CV) percentages. Some stages of the application are more predictable than others. All prediction methods predict the total runtime with a relative error of 10.55% at the worst and 8.42% at the best and a coefficient of variance of 87.17% and 90.74% respectively. The core of the application, namely, transferring the data, running the computation, and finally transferring the results, has a relative error of 11.33% at the worst and 7.92% at the best with a coefficient of variance of 98.15% and 106.65% respectively . (Note that, expectedly, all prediction methods have higher relative error for more stages with high variability.)

From the data, one can observe that the time series prediction method and the most accurate linear regression prediction method, namely, LRP with full history, have the smallest relative errors. This indicates that predicting the performance of an application by those of probes can be as good as predicting

by the history of the application itself. The hybrid method with full history pays a little cost of accuracy due to its online attribute because both of its TSP and LRP parts introduce errors. But it is still almost as accurate as the previous two methods. Methods with last 10 measurements have worse accuracy, which may seem counterintuitive. It appears that "ancient history" is of value in predicting future runs of the probe. Methods with fixed training set have the worst accuracy because the fixed linear function may not fully capture the relationship between the probe and the application. Still, with error rates of about 10%, these methods may be deemed preferable and are probably more practical given that they do not require the application to run periodically.

Overall, unless one's grid application spends most of its time in queues (an unfortunate attribute of the system) or an inordinate amount of time staging data on shared login nodes (an unfortunate design of the application), reasonably accurate forecasts of application performance can be obtained from benchmark probe measurements.

2.6 Related Work

This work is related to many previous projects that have produced benchmarks developed for microprocessors and for High Performance Computing (HPC) systems, and more recently for to Grid computing. Benchmarks for traditional microprocessors and HPC systems fall into at least two main categories. First are low-level "probes" that measure the rates at which a machine can perform fundamental operations. Examples of this class include MAPS [119, 78], STREAM [126], the Intel ® MPI Benchmark [62] (formerly Pallas PMB) and SKAMPI [117] MPI benchmarks, and to some extent the LINPACK benchmark[74]. In the specific context of probes for Grid platforms, one can find the GrASP project [34], which is described and used in this work, and other projects such as RGRBench [100] that measures the performance of Grid information service and was used in [69]. The second category of benchmarks are ones designed to capture the computational needs of a class of applications. In HPC, among the best known of these are

the NAS Parallel Benchmarks (NPB) [13] that are based on computational fluid-dynamics problems. Some other influential suites of this kind are SPEC [124], ParkBench [55] and SPLASH [138]. In the context of Grid computing, the NAS Grid Benchmarks (NGB) [47] have been developed as and extension to the NPB and other individual applications benchmarks have been developed such as the PreCo benchmark used in this work.

2.7 Conclusions and Future Work

A key challenge for improving the infrastructure and the operation of large-scale federated grid platforms is that of measuring and understanding resource availability and performance. This chapter has used the Inca system in conjunction with the GrASP benchmark probes and the PreCo application to monitor two state-of-the-art grids over a 6-month time period. The performance monitoring data has made it possible to quantify the performance variability that may be expected on such platforms and the sources of that variability. It has been found that, expectedly, wait times in batch schedulers' queues is the most relevant and prevalent source of performance variability. In terms of availability, the benchmarks have experienced rather low success rates in the experiments (in the 55%-80% range), showing that these platforms are still far from having what one would consider "good" availability. This work attempted to explain the sources of all failures, and found that some key failures came from the middleware infrastructure. Most of these failures should be easily correctable. Although it was not the focus of these experiments, it seems clear there is use for the probes as a diagnostic tool for platform administrators to troubleshoot the infrastructure.

With three different prediction methods, predictions for the application's total runtime were able to be made with a relative error of 10.55% at the worst and 8.42% at the best using the different methods. Importantly, some of these methods, after some training period, rely solely on the periodic execution of light-weight benchmark probes.

This chapter, in part, is a reprint of the material as it appears in the 7th

IEEE/ACM International Conference on Grid Computing (Grid'06), a joint work with Omid Khalili, Catherine Olschanowskyd, Allan Snavely, and Henri Casanova. The dissertation author was the primary investigator and author of this paper.

Table 2.7: Relative Error (and Coefficient of Variance) Percentages for Different Prediction Methods.

	Total Runtime (RT)	RT w/o Setup/ Cleanup	RT w/o Setup/ Cleanup /Queue	Stage Data	Build Executable	Actual Computation	Actual Queue
Average PreCo Measurement	816.61	715.45	608.07	33.50	17.42	312.89	107.37
Stdev	112.52	103.74	72.16	37.65	5.82	19.93	77.79
TSP (NWS)	8.57 (90.32)	8.47 (95.66)	7.92 (106.65)	29.44 (97.99)	3.84 (165.64)	4.39 (80.45)	29.37 (91.05)
LRP w/ Full History	8.42 (90.74)	8.73 (90.71)	8.08 (110.68)	27.62 (84.5)	4.86 (136.44)	5.83 (291.95)	33.8 (74.25)
LRP w/ Last 10 Measurements	9.13 (78.62)	9.35 (83.49)	9.44 (125.48)	67.89 (415.24)	6.29 (293.72)	11.95 (408.4)	28.31 (81.85)
LRP w/ Fixed Training Set	9.96 (90.61)	10.64 (90.24)	10.57 (92.33)	24.65 (78.9)	4.55 (140.17)	4.41 (85.34)	37.13 (76.33)
TSP+LRP w/ Full History	8.99 (85.5)	9.1 (87.69)	8.54 (107.93)	30.92 (82.14)	4.79 (130.75)	4.41 (75.72)	34.05 (84.04)
TSP+LRP w/ Last 10 Measuremetns	9.14 (81.84)	9.11 (87.4)	8.49 (97.04)	65.88 (227.4)	4.27 (121.68)	5.38 (105.91)	29.22 (83.44)
TSP+LRP w/ Fixed Training Set	10.55 (87.17)	11.11 (88.86)	11.33 (98.15)	28.43 (87.64)	4.69 (133.39)	4.55 (84.14)	37.84 (84.64)

Chapter 3

Automatic Recognition of Performance Idioms in Scientific Applications

In scientific applications, space, time, local and long-range interactions are represented by abstract programming constructs. Despite many different implementations, there are fundamental similarities in these abstractions: space is usually modeled by structured or unstructured grids, time is often modeled by the outermost iteration loop (time step), and interactions are modeled by systems of equations that are solved by iterative explicit or implicit methods.

As a result, a number of similar constructs, design patterns, and data flows can be found in various scientific applications [46, 68, 54]. Extraction, measurement and analysis of these representative constructs can result in a good understanding of the application performance and suggesting how best to map the applications to computer architectures. The performance analysis community has long realized this and utilized the common programming constructs and typical application kernels as benchmarks. But how to quantify the representativeness of benchmarks remains a hard problem. Usually the selection of benchmarks is to some extent subjective and decided by human sense. This research found that identification of the common constructs or data flow patterns, such as stream, transpose, reduction, random access and stencil, in real applications, along with their coverage

coefficients, would allow us to use these constructs as performance proxies for real applications quantitatively and allow us to derive application requirements for new platforms based on the analysis of a few simple constructs.

Understanding these program constructs helps not only new hardware design not also performance optimizations. For example, reduction is one of the program constructs well studied by the compiler communities. Quite a few compilers [76, 108, 51] implemented advanced reduction recognition for program optimization. Furthermore, hybrid hardware is emerging with different components designed for different program constructs. RoadRunner [15] is the first petaflops supercomputer with hybrid hardware: Opteron ® cores plus Cell ® processors. GPU computing is becoming more and more important. The first place of the current Top500 list [133] is GPU machine, Tianhe-1A. How to recognize suitable program constructs for GPU execution automatically will be critical for the usability and efficiency of these GPU machines. FPGA machines will have even more diversities. One example is Convey ® HC-1 [22], which provides a handful application-specific co-processors called personalities for Intel ® processors. To explore the potential of this kind of hardware needs automatic recognition of different program constructs (personalities) and mapping them to corresponding hardware components. In one of our un-published works, we were able to recognize the Gather/Scatter constructs in real applications and send it to the Convey FPGA accelerator thus speeding as much as 20x.

On the road to exascale supercomputing, performance is not the only concern. In fact, power consumption is becoming the limit for scalability. To solve the problem, our colleague Professor Andrew Chien has proposed a new paradigm called 10x10 [33]. The basic idea is to integrate 10 different specialized cores onto one chip and use the best matched ones for specific program constructs. The recognition technique can be used to recognize these program constructs and map them to corresponding cores.

These similar constructs are called **performance idioms** or, for simplicity, just **idioms**. Formally, idioms are primitive program components, which each capture a pattern of computation and communication over arrays or sub-arrays

common in applications. Some questions about the idioms are: how many idioms are needed to cover most application codes? Can these idioms approximate the performance of real applications? And is it possible to find out these idioms from application codes automatically? These are the questions explored in this chapter. To this end, this work designs and develops a static analysis tool to recognize idioms automatically, and then verifies the tool by manual analysis. Also the performance of the idiom benchmarks with their corresponding instances in application codes are compared. The NAS Parallel Benchmark (NPB) [13] suite will be used as a case study. This work will be limited to Fortran code. Extending to other languages will be one of the future works.

The rest of the chapter is organized as follows. In the next section, five well-studied idioms, that is, Stream, Transpose, Random access, Reduction, and Stencil are defined. Then section 3.2 proposes a compiler-based method to recognize these idioms within real application codes automatically and presents an implementation based on the open source compiler Open64. The experimental results of the prototype system applied to the NPB will be described in section 3.3. Also the code coverage and performance approximation results of NPB will be presented. Section 3.4 includes some related work in the field. Conclusions and future work will be given in the last section.

3.1 Definitions of Five Idioms

Idioms represent basic operations of applications. Systems used for scientific computations should deliver high performance for all, or at least the dominant part of, the idioms. The complete list, or even just a dominant list of idioms, has not been identified yet. However, some idioms such as Stream, Transpose, Random access, Reduction and Stencil are already well studied in the community. This section formally defines these five idioms by the concept of affinity relation and dependence.

One can write an array variable assignment of array B to array A inside a

loop nest using pointer arithmetic:

$$A + V \cdot I = B + W \cdot I,$$

where I is the vector of loop variables $(i, j, k \ldots)$ and V and W are vectors expressing affinity. The matrix $(V, W)^T$ is called the **affinity relation** between A and B. In another word, if variables A and B are on the left hand side (LHS) and the right hand side (RHS) of an assignment, respectively, the affinity relation between A and B is a matrix composed of the coefficients of their indexes if they are array variables or just 0's otherwise. If a term including an index is not linear, its coefficient is marked as $FUNC$. For example, the assignment $a(i, j) = b(c(j), i)$ can be transferred to its linearized form $a(i + n * j) = b(c(j) + m * i)$, where n and m are the sizes of the leading dimensions of a and b, respectively. Then the affinity relation between a and b can be represented by:

$$\begin{pmatrix} 1 & n \\ m & FUNC \end{pmatrix}$$

where the first row is for a and the second is for b. The first column is for loop variable i and the second one is for j. In general, an affinity relation A looks like:

$$\begin{pmatrix} a_{11} & a_{12} & \ldots \\ a_{21} & a_{22} & \ldots \end{pmatrix}$$

where the first row is for the LHS variable and the second one is for the RHS variable.

Dependence, or data dependence, is a constraint between statements to guarantee the order in which data are produced and consumed. Dependence information can be easily derived from a compiler's intermediate representation, such as a dependence graph [4, 87]. A dependence cycle is a directed cycle in the dependence graph.

With the above two definitions, one can define the five idioms as follows. One thing worthy of mention is that every definition here is with respect to a specific surrounding loop. More details will be given in section 3.2.4.

Definition 1 (Stream) *An assignment is classified as Stream if*

- *for each affinity relation A between LHS and an array variable on RHS, there do not exist two loop indexes i and j such that $a_{1i} < a_{2i}$ but $a_{1j} > a_{2j}$;*
- *there is no FUNC element in any affinity relation;*
- *it is not involved in a dependence cycle.*

As follows is the triad version of the Stream benchmark designed by McCalpin [126]. If its data set size is big enough, a Stream idiom will consume the full bandwidth between the processor and the main memory.

```
do i=1, m
  a(i) = b(i) + const * c(i)
end do
```

Definition 2 (Transpose) *An assignment is classified as Transpose if*

- *for each affinity relation A between LHS and an array variable on RHS, there exist two loop indexes i and j such that $a_{1i} < a_{2i}$ but $a_{1j} > a_{2j}$.*

Here is a typical example of the Transpose idiom. It reads a matrix by rows and writes it by columns, which will challenge the strided access performance of the memory system.

```
do i=1, m
  do j=1, n
    a(j,i) = b(i,j)
  end do
end do
```

Definition 3 (Random Access) *An assignment is classified as Random access if*

- *for each affinity relation A between LHS and an array variable on RHS, there exist at least one element marked as FUNC (it is understood that, in general, FUNC **could** represent a simple access pattern, but the assumption of random is still useful for performance classification).*

Here is an example of Random Access idiom, which reads unpredictable memory locations and writes them into contiguous memory locations. Its performance is limited by memory latency.

```
do i=1, m
  a(i) = b(c(i))
end do.
```

Definition 4 (Reduction) *An assignment is classified as Reduction if*

- *the LHS variable appears on the RHS with the same subscript if any;*
- *for each affinity relation A between LHS and an array variable on RHS, if* $a_{1i} \neq 0$*, there must be* $a_{2i} \neq 0$*;*
- *the first level operators of RHS are associative;*
- *it is involved in dependence cycles.*

A typical one-dimensional Reduction idiom looks like as follows, which reduces an array by an associative operation. Its performance can be optimized by vectorization.

```
do i=1, m
  s = s + a[i]
end do
```

Definition 5 (Stencil) *An assignment is classified as Stencil if*

- *for each affinity relation A between LHS and an array variable on RHS, there do not exist two loop indexes* i *and* j *such that* $a_{1i} < a_{2i}$ *but* $a_{1j} > a_{2j}$*;*
- *it is involved in dependence cycles.*

Below is a typical one-dimensional Stencil idiom. Each update of an array element depends on the values of its neighbors. It tests the performance of instruction scheduling.

```
do i=1, m
  a(i) = a(i-1) + a(i+1)
end do
```

Sometimes more than one idiom definitions are satisfied. Then the assignment is classified as a hybrid idiom. For example, the Sparse-Matrix-Vector multiplication (SMV) as follows is a hybrid idiom composed of Reduction and Random.

```
do j=1, lastrow-firstrow+1
  sum = 0.d0
  do k=rowstr(j), rowstr(j+1)-1
    sum = sum + a(k)*p(colidx(k))
  enddo
  q(j) = sum
enddo
```

3.2 Automatic Recognition by Compiler

Previous section has defined five well-studied idioms. It is unknown if it is feasible to use such a small number of idioms to completely or mostly represent certain scientific application codes and if these idioms are useful as performance proxies of real applications. This section is proposing a compiler-based static analysis method to extract idioms from real application codes automatically. A prototype implementation will also be described in the last sub-section.

The method includes four stages. First, extract the loop nest structure from the compiler intermediate representation to construct the Loop Nest Graph (**LNG**). And then traverse the loop nest to scan all the assignments and extract their affinity relations to construct Affinity Relation Graph (**ARG**). By removing temporary variables, the Reduced Affinity relation Graph (**RAG**) will be built. Finally, idioms are recognized by matching the RAG with idiom definitions. The details of these four stages will be introduced in the rest of the section. At the

same time, the whole analysis process of the example Fortran routine shown in Figure 3.1 will be gone through.

```
subroutine foo
integer i, j
do j = 1, 100
  do i = 1, 100
    tmp = a(i,j) - b(i)
    sum(i) = sum(i) + tmp*tmp
  enddo
enddo
return
end
```

Figure 3.1: A complete example of Fortran routine

3.2.1 LNG: Loop Nest Graph

The Loop Nest Graph (LNG) is used to represent the control flow structure of the whole routine, especially the loop nest structure. Its goal is to provide context information for the following analysis such as surrounding loops, loop indexes and loop-carried dependences. This work does not consider improper regions, which are multiple-entry strongly connected components, or loops sharing the same header. Figure 3.2(a) shows an example Control Flow Graph (CFG) [4, 87] of improper regions. And an example CFG of loops with the same header is shown in Figure 3.2(b).

A LNG is a directed tree. Its root represents the whole routine and other nodes stand for natural loops or other control structures such as if-statements. There is a directed edge between two nodes if and only if the control flow construct of the parent node includes that of the child node. Usually, the LNG can be easily built from the compiler intermediate representation such as Control Flow Graph (CFG) or its intermediate language if it is designed to have the control flow

(a) Improper region (b) Loops with the same header

Figure 3.2: Example CFGs of improper region and loops with the same header

structure embedded like the Open64 High WHIRL that will be introduced in the next section. Figure 3.3 shows the LNG of the example in Figure 3.1.

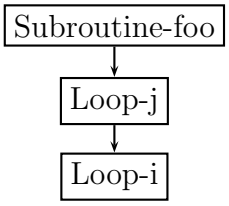

Figure 3.3: LNG of the example in Figure 3.1

3.2.2 ARG: Affinity Relation Graph

With the LNG in hand, one can traverse the whole control flow structure recursively to scan all the assignments and build Affinity Relation Graphs (ARG) for them. To obtain context information easily, each ARG is hung under its corresponding control flow construct in the LNG.

An ARG itself is also a directed tree graph. Its root represents the variable on LHS, which equals the whole expression on RHS, and other nodes stand for the sub-expressions (or variables on leaf nodes) on RHS. There is a directed edge between two nodes if and only if the expression of the parent node includes that of the child node. The affinity relation matrix rows are distributed across the corresponding variable nodes. Since data might flow from one assignment to another by temporary variables, a method similar to Static Single Assignment (SSA) [39] is adopted to connect these assignments together to be a so-called **combined assignment**. Specifically, a version number is attached to each variable. If it is a

scalar variable, its number is incremented only while it is on LHS. Being an array variable, its number is incremented every time it appears in an assignment in order to keep the subscript information for each instance. This work does not consider data flows across borders of control flow constructs and so phi-nodes [39] are not added.

Figure 3.4 shows the ARG of the example in Figure 3.1. Here the extensions of variable names stand for version numbers. Since only incrementing the version number of a scalar variable when it is on LHS, all three instances of the variable *tmp* have the same version number and they are represented by a single node in the ARG. As a result, the two assignments are connected together and finally recognized as a single idiom instance. In contrast, the instances of the same array variable *sum* have different version numbers and they are represented by different nodes in the ARG. In fact, array variables are used as the borders of combined assignments.

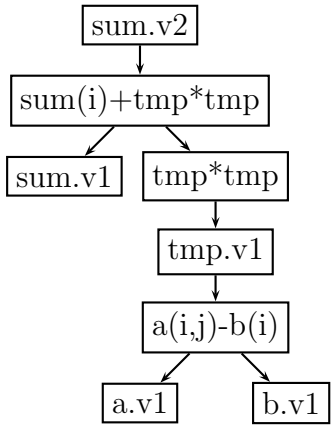

Figure 3.4: ARG of the example in Figure 3.1

3.2.3 RAG: Reduced Affinity relation Graph

Though affinity relation between variables have been caught in the ARG, it is still not enough for idiom recognition because there are temporary variables in the middle to interfere. It is needed to remove these and reduce the ARG into a Reduced Affinity relation Graph (RAG). A RAG is still a directed tree graph.

Instead of deleting the nodes directly from the ARG, it is chosen to build a new graph by copying the root and the leaves from the ARG and connecting them accordingly. By keeping the ARG intact, one can turn back to the original graph in case he needs more information. The affinity relation matrix rows are still distributed across the variable nodes. Figure 3.5 shows the RAG of the example in Figure 3.1. Now we can easily see that the LHS of the combined assignment is $sum.v2$ and the RHS includes $sum.v1$, $a.v1$ and $b.v1$. The next part will analyze the affinity relations between $sum.v2$ and the RHS variables to match the idiom patterns.

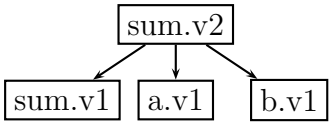

Figure 3.5: RAG of the example in Figure 3.1

3.2.4 Idioms Recognition

Having a RAG for each combined assignment, also being able to obtain dependence information from the compiler dependence graph easily, one can check each RAG to see if it satisfies any idiom definitions given in the section 3.1 and classify the assignment as the matched idiom. Since both affinity relations and dependence cycles are related to the surrounding loops, an idiom has to be classified with respect to a specific surrounding loop. To this end, an algorithm similar to Allen and Kennedy's vectorization algorithm [66] is adopted to classify idioms from the outer most loop to the inner ones. For a given loop, only the dependence cycles whose maximum levels equal the current loop are considered. Here **level** means the index of the leftmost non-equal element of the dependence direction vector [66]. Specifically, this work only keep the dependences whose levels are higher than the current loop, that is, the loop carried dependences over the loops inside and including the current loop plus the loop independent dependences. Next the maximal Strongly Connected Components (SCC) of the dependence graph is

constructed. During the construction, the maximum level of each SCC is recorded. While being matched with the idiom definitions, an assignment is classified as "in dependence cycle" if and only if it belongs to an SCC whose size is more than one and the maximum level equals the current loop, or it has a self-dependence carried by the current loop.

Let us continue to analyze the example in Figure 3.1. As you may observe, over the j loop, there is a self-dependence for sum, from $sum.v2$ to $sum.v1$. Since the assignment also satisfies the other requirements of the Reduction definition, it is classified as Reduction with respect to the j loop. Now we come to the i loop. Since the dependence over the j loop is removed, there is no dependence cycle on the assignment at this point. The assignment also satisfies the other requirements of the Stream definition and is classified as Stream with respect to the i loop.

3.2.5 Implementation on Open64

The Open64 compiler was originally developed by SGI ® as MIPSpro, and was open-sourced later. It also benefited from the Open Research Compiler (ORC) [97] project by Intel ® and the Chinese Academy of Sciences, which focused on IA64 architecture. Open64 is an industrial-strength compiler and contains an advanced and complete optimization framework, including scalar optimization (WOPT), loop nest optimization (LNO), inter procedural optimization (IPA) and code generation (CG). Its intermediate language called WHIRL was designed to have five levels (Very High, High, Mid, Low and Very Low) to fit the different requirements of different analyses and optimizations. The prototype was implemented in the LNO phase of the High WHIRL level. In this phase, the high level control flow constructs such as loops and if-statements are preserved and array structures are kept, which aids the implementation, particularly the construction of the LNG. The LNO phase also provides the facilities for dependence analysis, which is essential for the implementation.

3.3 Experiment Results

The previous section has described a 4-stage method and its implementation for automatic idioms recognition. This section will verify the accuracy of the prototype system. Also it is interesting to check if a small number of idioms can cover most scientific codes and approximate the performance of real applications To this end, this work starts with a manual analysis of code coverage on the NPB codes. And then, the result of the manual analysis is used to verify the prototype system. Finally, compare the performance of the idiom benchmarks with their corresponding instances in the NPB codes.

3.3.1 The NAS Parallel Benchmark

This section uses the NAS Parallel Benchmark (NPB) suite [13] as a case study. NPB is a set of benchmarks designed by NASA Advanced Supercomputing division (NAS) to evaluate the performance of supercomputers. The benchmarks are derived from Computational Fluid Dynamics (CFD) applications. The reasons to choose NPB are two fold. First, it was originally developed in 1990's and has been well studied and widely used by the scientific computing community [130, 79, 93, 137, 32]. In a recent Berkeley technical report [10], the authors found that most of the important modern numerical methods have their corresponding NPB benchmarks. Second, the code size of NPB is reasonable for manual analysis of code coverage, whose results are necessary to verify the automatic recognition tool.

NPB includes five kernels (EP, MG, CG, FT, and IS) and three pseudo applications (BT, LU and SP). Since EP (representative of input generation) is not a numerical, array traversing, application, which is theorized the idioms may cover, and IS is written in the C language, this work only applied the analysis to the other six programs.

3.3.2 Code Coverage

As mentioned above, it will be helpful to understand how many idioms are needed to cover most application codes; specifically, whether a small number of

idioms, say, the above five idioms can cover most application codes. To answer the question, a good way is to analyze some typical scientific applications to find out all the idioms and their **code coverage**, that is, the percentages of the static code classified as idioms (dynamic code coverage will be the next step). This work started with analyzing the NPB codes to prove the feasibility. The manual analytical results presented here will also act as the verification of the automatic tool later.

Table 3.1: Static breakdown of the NPB by idioms

	MG	CG	FT	BT	SP	LU
Stream	76	43	19	501	283	390
Transpose	0	0	2	3	3	0
Random	18	3	1	0	0	0
Reduction	2	7	0	7	47	4
Stencil	2	4	0	0	45	5
SMV	0	2	0	0	0	0
Total	98	59	22	511	378	399

The analysis results show that the above five idioms suffice to cover all the assignments of the six NPB programs within loops. Table 3.1 shows the static breakdown of the NPB by the five idioms. There are two examples worthy of mention here. The subroutine $Swarztrauber()$ in FT can be viewed as a "shuffle" on the function level. But the analysis proceeds on the statement level and all the assignments of the subroutine are classified as Stream. Another example is the Sparse-Matrix-Vector multiplication (SMV) in CG. As mentioned above, it is classified as a hybrid idiom composed of Reduction and Random. For clarity, it is still shown separately in the table. Though real application codes might be much more complicated and versatile than the NPB, the results verified that it is feasible to use a small number of idioms to cover application codes.

3.3.3 Prototype Verification

To verify the prototype system, this work applied it to the NPB and compared the results with those of the manual analysis. The results are shown in Table 3.2. The column "Number of mis-recognized assignments" shows the number of differences between the automatic and manual analyses. And the column "Relative error" shows the ratio of the above number to the total number of assignments. We can see that the highest error is 10.2%. The results show that automatic idiom recognition is feasible.

Table 3.2: Automatic idioms recognition for NPB

Benchmarks	Total number of assignments	Number of mis-recognized assignments	Relative error
MG	98	2	2.0%
CG	59	6	10.2%
FT	22	2	9.1%
BT	511	5	1.0%
SP	378	21	5.6%
LU	399	13	3.3%

3.3.4 Performance Approximation

By definition, idioms represent the basic components of scientific applications. But what can be said about their performance? Can one use the performance of idiom benchmarks to approximate the performance of idiom instances in real application codes? If yes, it will be easier to extract application requirements for new systems and allow compiler and hardware designers to focus on the performance of a few simple idioms rather than on full applications.

To investigate the question, this work developed a set of idiom benchmarks and tried to use their measured performance to approximate the performance of idiom instances in the NPB programs. For simplicity, each idiom benchmark was written in a single typical form of the idiom. For example, though there are several

Table 3.3: Configurations of the experiment platforms

	Itanium2	Power4
Frequency	1.5 GHz	1.7 GHz
L1 Instruction Cache	16 KB	64 KB
L1 Data Cache	16 KB	32 KB
L2 Data Cache	256 KB	1.5 MB
L3 Data Cache	6MB	128 MB
Main Memory	4 GB	32 GB

versions of Stream idiom in McCalpin's Stream benchmark [126], only the triad version is adopted in the experiment. Also, in each NPB benchmark, only the performance-dominant instances are considered. For a specific idiom benchmark, there may be quite a few corresponding instances. The benchmark was scaled across a large enough range of data set sizes to cover those of all the corresponding instances in the NPB codes. The benchmarks and the NPB codes were measured on two different platforms: Intel ® Itanium2 [64] processor and IBM ® Power4 [109] processor. Table 3.3 shows the detailed configurations of these two platforms.

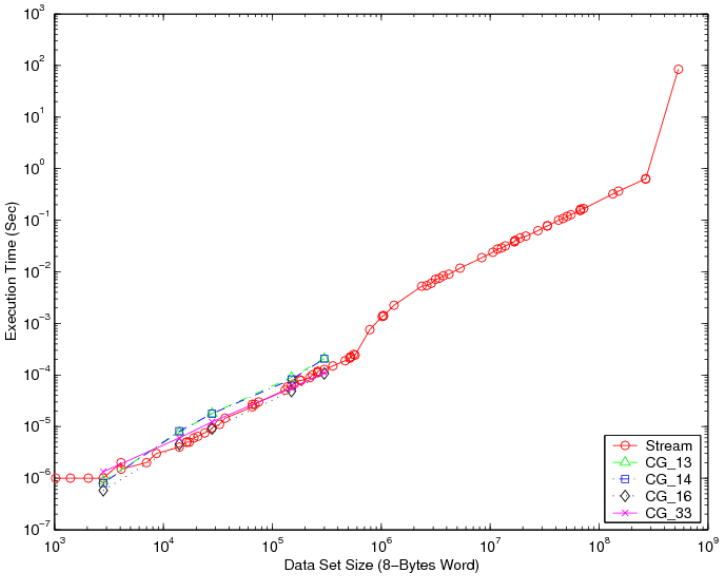

Figure 3.6: Execution time versus data set size of the Stream idiom benchmark and its instances in CG on Itanium2

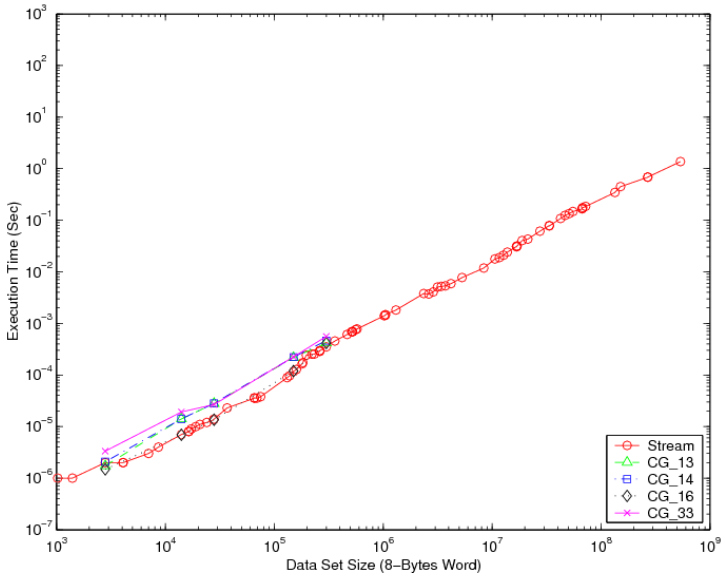

Figure 3.7: Execution time versus data set size of the Stream idiom benchmark and its instances in CG on Power4

The average difference between the idiom benchmarks and their instances in the NPB on Intanium2 is 30.2% and that on Power4 is 36.8%. To make it more intuitive, it is shown here the comparison between the Stream benchmark and its corresponding instances in CG on the two platforms in Figure 3.6 and Figure 3.7, respectively. CG_xy's in the figures are the Stream instances in CG. As referred to above, since each idiom benchmark scales across a large enough range of data set sizes to cover those of all the corresponding instances in the NPB codes, the data set size range of a specific instance is usually smaller than that of its corresponding idiom benchmark as shown in the figures. From the curves we can see that the Stream benchmark and its corresponding instances in CG have similar trends, which also happens between other idioms and NPB programs.

The relative matching between the idioms and their instances were also investigated. The best predictor of an instance is the idiom benchmark with minimum performance difference from the instance. If the best predictor of an instance is its corresponding idiom classified by the static analysis, it is a hit. Otherwise, it is a miss. The investigation was applied to the performance-dominant instances measured. The hit rate on Itanium2 is 79.3% and that on Power4 is 48.3%, which

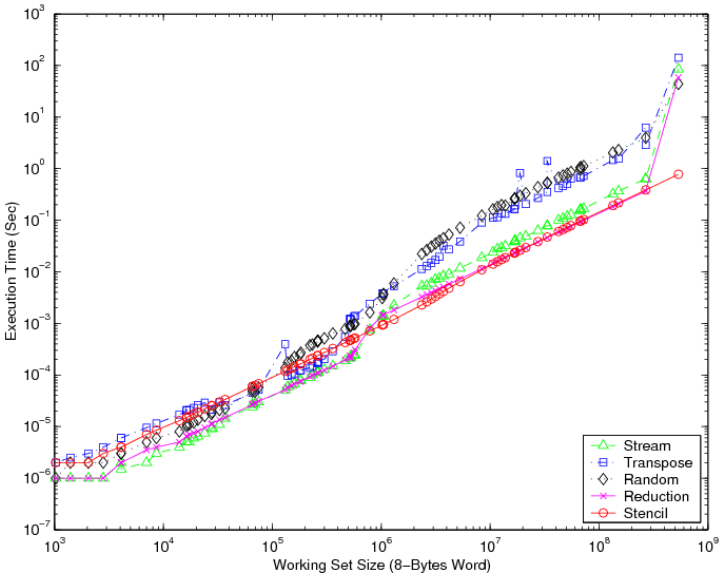

Figure 3.8: Execution time versus data set size of the idiom benchmarks on Itanium2

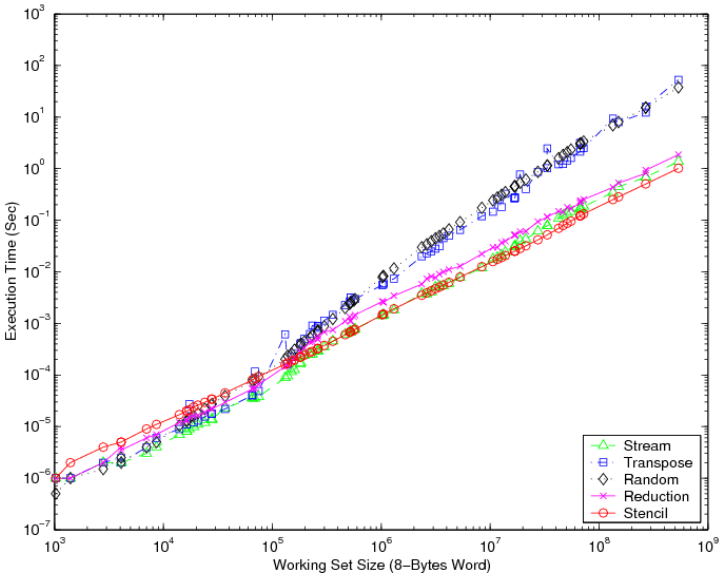

Figure 3.9: Execution time versus data set size of the idiom benchmarks on Power4

are not satisfying. It is because different idioms might have similar performance and interfere with each other though they have different data flow patterns. As shown in Figure 3.8 and Figure 3.9, the performance trend of Transpose is quite

similar to that of Random, and the trends of the other three idioms are similar. Observing this leads to grouping the idioms with similar performance together and applying the investigation again. The hit rates are much improved: 96.6% on Itanium2 and 79.3% on Power4.

This section showed both the absolute and the relative performance matchings between idioms and their corresponding idioms in the NPB code. To improve the absolute performance matching, it might be necessary to consider idiom variations, e.g. the same idiom with different memory access strides. In contrast, for relative matching, it will be better to group the idioms with similar performance together and have less categories. The work will continue on these two directions to explore the trade-off between them. The performance matching between the idioms and their instances in the NPB codes shows that it is feasible to use simple idioms to approximate the performance of real applications.

3.4 Related Work

This work is based on many successful investigations. This section will survey some related works in different fields.

3.4.1 Benchmarks and Application Requirements

This work is not the first to propose the idea to use common programming constructs as benchmarks or application requirements. The HPC Challenge (HPCC) benchmark suite [56] was developed by the University of Tennessee, Knoxville, to measure and evaluate the performance of different components, such as processor, memory and interconnection, of high performance computers. It consists of 7 tests, ranging from low level idioms such as Stream, Parallel Matrix Transpose (PTRANS) and RandomAccess to high level kernels such as High Performance Linpack (HPL) and FFT. Though the HPCC test set has overlap with the idiom set, the idioms are all on the same low level and more general in application codes.

The Seven Dwarfs project [10] from the University of California, Berkeley

tried to identify a number of basic numerical methods such as N-body method, structured and unstructured grids, and MapReduce, which are important for science and engineering applications and representative for application requirements. Dwarfs are algorithmic methods whose levels are much higher than those of our idioms. They can even be the highest level algorithm of a large application. The high level of abstraction allows reasoning about the Dwarfs' behavior across a broad range of applications. But they are not easy to be recognized automatically.

3.4.2 Archetypes and Kernel Coupling

Parallel programming archetype [112] is an abstraction composed of computational structure, parallelization strategy and implied patterns of data flow and communication, proposed by researchers from California Institute of Technology for parallel program developing and performance analysis. A program developed using archetype includes two parts: archetype-specific communications and application-specific computations. To analyze its performance, one need to measure the execution time of these two parts respectively, and then build a performance model or simulation according to the computation and communication structure implied by the archetype to calculate the whole execution time. Though the compositional performance analysis idea of the archetype is similar to ours, its granularity is much larger than us. Furthermore, it is a manual method and a programmer has to understand the program structure to decompose the program and build the performance model or simulation by hand.

The Prophesy project [131] from Texas A&M University has a similar idea of compositional performance analysis dividing a large program into small kernels to analyze. Since significant works has been done on performances of small kernels, the project focuses on kernel coupling [130], the performance relations between different kernels in a single program. By this way, performance predictions can be greatly improved over the traditional technique of just simply summing up the execution times of the individual kernels in a program. Again, comparing with this work, the granularity of kernel coupling is larger, usually on the function level, and the application decomposition (during database building) is manual.

The kernel coupling idea is interesting and may be also necessary for this work if later aiming at a higher goal of performance prediction instead of just defining application requirements. However, it will be difficult if not impossible to apply kernel coupling directly on the fine-grain idioms. Some variations are probably needed.

3.4.3 Machine Idioms

Machine idioms [4] are special instruction sequences common in assembly codes. Such a sequence usually comes from a specific high-level operation (e.g., increment of a variable). Though the operation is considered by the programmer as atomic, it might be translated into a sequence of instructions on a target machine. If such an operation is common and the sequence repeats frequently, one might want to add a new instruction (e.g., instruction INC) into the ISA of the target machine and then a compiler can replace the whole sequence with a single instruction. There are quite a few works [9, 113] in the community trying to recognize high-coverage machine idioms for optimal ISA extensions or other architecture designs. Usually they also use graph matching to recognize idioms. However, these works are quite different from ours. First, a machine idiom is defined as an instruction sequence in dynamic context. It does not consider iterative behaviors over loops and sometimes is even limited in a basic block. Second, since machine idioms are specific instruction sequences, they only need simple **exact** graph matching to recognize.

3.4.4 Reduction Recognition

Reduction recognition is well studied in compiler research, especially in parallelizing compilers. The HPF compiler [76] from Rice University, the Polaris compiler [108] from University of Illinois and the SUIF compiler [51] from Stanford University are three of such systems. All of them apply dependence analysis and pattern matching methods to recognize reductions. The SUIF compiler can even recognize and parallelize inter-procedural and sparse reductions. However,

their goal is for optimization instead of workload characterization and performance analysis. They usually focus on only a few of the idioms that can be optimized, especially reductions. Even in these considered idioms, they avoid some complicated forms to avoid errors or expensive optimization costs. In contrast, this work is more general and tries to catch as many idioms as possible.

3.5 Conclusions and Future work

By definition, performance idioms are the basic components of scientific applications. This chapter proposed an automatic idioms recognition method and implemented the method, based on the open source compiler Open64. Applied to the NAS Parallel Benchmark (NPB), the prototype system achieved an accuracy of more than 90% compared with a manual idioms classification. It was also found that five idioms suffice to cover 100% of the six NPB codes (MG, CG, FT, BT, SP and LU) and the performance approximation between the idioms and their corresponding instances in the NPB codes is up to 96.6% accurate. The automatic recognition method and prototype system enable people to find out the representative idioms of real scientific applications automatically. The preliminary results proved to some extend that a small number of idioms can cover most scientific codes and approximate the performance of real applications.

With the verified automatic tool, it is possible to prove the code coverage hypothesis by examining more and larger application codes such as CPU2006 [37]. This work is limited to static code coverage. One of the next steps is to calculate the dynamic code coverage by combining the static results with the dynamic profile information. With dynamic code coverage, one can approximate the application performance automatically and check more application codes to test the hypothesis about performance approximation. It is in progress to apply the technique for performance optimization on GPU and FPGA machines. Moreover, according to previous research on non-volatile memory [53, 27, 52, 83], performance behaviors on these new storage technologies are totally different from traditional spinning disks. As a result, similar ideas can be applied for hybrid storage systems.

This chapter, in part, is a reprint of the material as it will appear in the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), a joint work with Allan Snavely, Rob Van der Wijngaart, and Michael Frumkin. The dissertation author was the primary investigator and author of this paper.

Chapter 4

DASH: a Flash-based Data Intensive Supercomputer

Certain domains of science, such as genomics [35] and astronomy [144], are literally "drowning in a sea of data" in that disks are filling up with raw data from sequencing machines and space telescopes faster than that data can be analyzed. Some data analysis problems can be solved by parallel processing with many compute nodes thus spreading out the data across many physically distributed memories. Others, limited by low parallelism or challenging access patterns depend on fast I/O or large fast shared memory for good performance.

By talking to users, examining their applications, and participating in community application studies [123][122][70][120], data intensive HPC applications were identified spanning a broad range of science and engineering disciplines that could benefit from fast I/O and large shared memory packed onto a modest number of nodes; included are applications in the growing areas of 1) data mining and 2) predictive science used to analyze large model output data.

In a typical *data mining* application, one may start with a large amount of raw data on disk [129]. In the initial phase of analysis, these raw data are read into memory and indexed; the resulting database is then written back to disk. In subsequent steps, the indexed data are further analyzed based upon queries, and the database will also need to be reorganized and re-indexed from time to time. As a general rule, data miners are less concerned about raw performance

and place higher value on productivity, as measured by ease of programming and time to solution [48]. Moreover, some data mining applications have complex data structures that make parallelization difficult [11]. Taken together, this means that a large shared memory and shared memory programming will be more attractive and productive than a message passing approach for the emerging community of data miners. I/O speed is also important for accessing data sets so large that they do not fit entirely into DRAM memory.

A typical *predictive science* application may start from (perhaps modest) amounts of input data representing initial conditions but then generate large intermediate results that may be further analyzed in memory, or the intermediate data may simply be written to disk for later data intensive post-processing. The former approach benefits from large memory; the latter needs fast I/O to disk. Predictive scientists also face challenges in scaling their applications due to the increasing parallelism required for peta-scale and beyond [11]; they benefit from large memory per processor as this mitigates the scaling difficulties, allowing them to solve their problems with fewer processors.

With forecasting the characteristics of data intensive applications in the future, it is found that today's supercomputers are, for the most part, not particularly well-balanced for their needs. Creating a balanced data intensive system requires acknowledging and addressing an architectural shortcoming of today's HPC systems.

The deficiency is depicted graphically in Figure 4.1; while each level of memory hierarchy in today's typical HPC systems increases in capacity by 3 orders of magnitude, the costs of each capacity increase are latencies that increase and bandwidths that decrease by at least an order of magnitude at each level. In fact, today's systems have a *latency gap* after main memory. The time to access disks is about 10,000,000 processor cycles - five orders of magnitude greater than the access time to local DRAM memory. It is almost as though today's machines are missing a couple of levels of memory hierarchy that should read and write slower than local DRAM but orders of magnitude faster than disk. Since some data sets are becoming so large they may exceed the combined DRAM of even large parallel

supercomputers, a data intensive computer should, if possible, have additional levels of hierarchy sitting between DRAM and spinning disk. To fill these missing levels, a data intensive architecture has at least two choices: 1) aggregate remote memory and 2) faster disks. A system named DASH was designed to make use of both. With these two additional levels (depicted in the Figure 4.1 as Remote Memory and Flash Drives), this work managed to fill the latency gap and to present a more graceful hierarchy to data intensive applications.

Section 4.1 describes the high-level design of DASH and compares its efficiency to other designs in the same space. Section 4.2 supplies the detailed "recipe" used to design and tune the high performing flash-based I/O nodes of DASH - the intent is that the description is detailed enough so that anyone can understand the design choices and duplicate them. Section 4.3 describes the performance of some scientific applications - the experiments showed that DASH can achieve up to two-orders-of-magnitude speedup over traditional systems on these data intensive applications. Section 4.4 discusses flash generally and lessons-learned. Section 4.5 is related work.

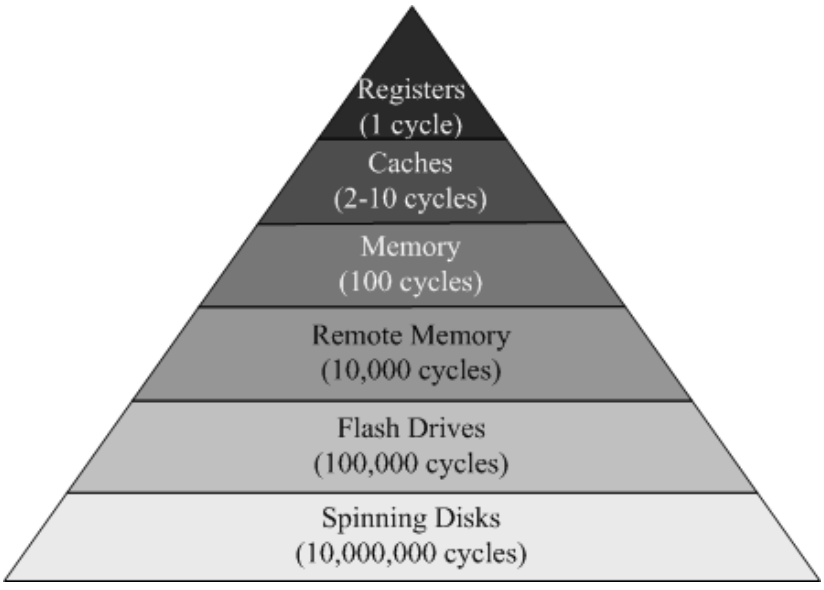

Figure 4.1: The memory hierarchy. Each level shows the typical access latency in processor cycles. Note the five-orders-of-magnitude gap between main memory and spinning disks.

4.1 System Overview

DASH is comprised of 4 "supernodes" connected by DDR Infiniband. Each supernode is physically a cluster composed of 16 compute nodes and 1 I/O node, virtualized as a single shared memory machine (see Figure 4.2) by the vSMP system software from ScaleMP ® Inc. [114]. Each compute node comprised of 2 Intel ® quad-core 2.4GHz Xeon Nehalem E5530 processors with 48GB of local DDR3 DRAM memory. As a result, each supernode has 128 cores, 1.2TFlops of peak capability, and 768GB of global (local + remote) shared memory. The I/O node is loaded with 16 Intel ® X25-E 64GB flash drives, which amount to 1TB in total capacity. DASH has 4 such supernodes in all, 64 compute nodes with 4.8 TFlops, 3 TB of DRAM and 4 TB of flash. DASH is a prototype of the larger National Science Foundation (NSF) machine code-named Gordon slated for delivery in 2011, which will have more (32) and larger (32-way) supernodes and will feature 245 TFlops of total compute power, 64 TB of memory, and 256 TB of flash drives.

4.1.1 Storage hierarchy

Flash drives provide the first level (closest to the level of spinning disks) to fill the latency gap. NAND Flash is a lively research and industry topic recently [31][3][84][98][50]. Unlike traditional electromechanical hard disks, flash drives are based on solid-state electronics and have quite a few advantages over hard disks, such as high mechanical reliability, low power consumption, high bandwidth, and low latency. Their latency is about 2 orders of magnitude lower than that of spinning disks. With these faster drives, it is possible to bring user data much closer to the CPU. Flash drives can be classified as MLC (Multi-Level Cell) and SLC (Single-Level Cell) drives. SLC was chose for longer lifetime, lower bit error rate, and lower latency. The prototype system DASH has 1 TB of flash drives per supernode (4 TB in all). Gordon will get more (8 TB per supernode, 256 TB in all).

Though flash drives are much faster than spinning disks, there is still a

big latency gap between DRAM memory and flash drives (see Figure 4.1). DASH is equipped on each compute node with 48GB of local DDR3 DRAM memory, that is, 6GB per core. In contrast, most existing supercomputers have only 1 to 2GB per core. So DASH already has a better ratio of DRAM to compute power - suitable for data intensive computing. Furthermore, as the second layer of latency-gap filler, the vSMP software is exploited to aggregate distributed memory into a single address space. That means every single core in the supernode can access all 768GB of (local + remote) memory possessed by (all 16 compute nodes of) one supernode. With such a large shared memory, users can deal with applications with large memory footprint but limited parallelism, or just use all that memory as a RAM disk for fast I/O. Users with less than 3/4 of a TB of data can move

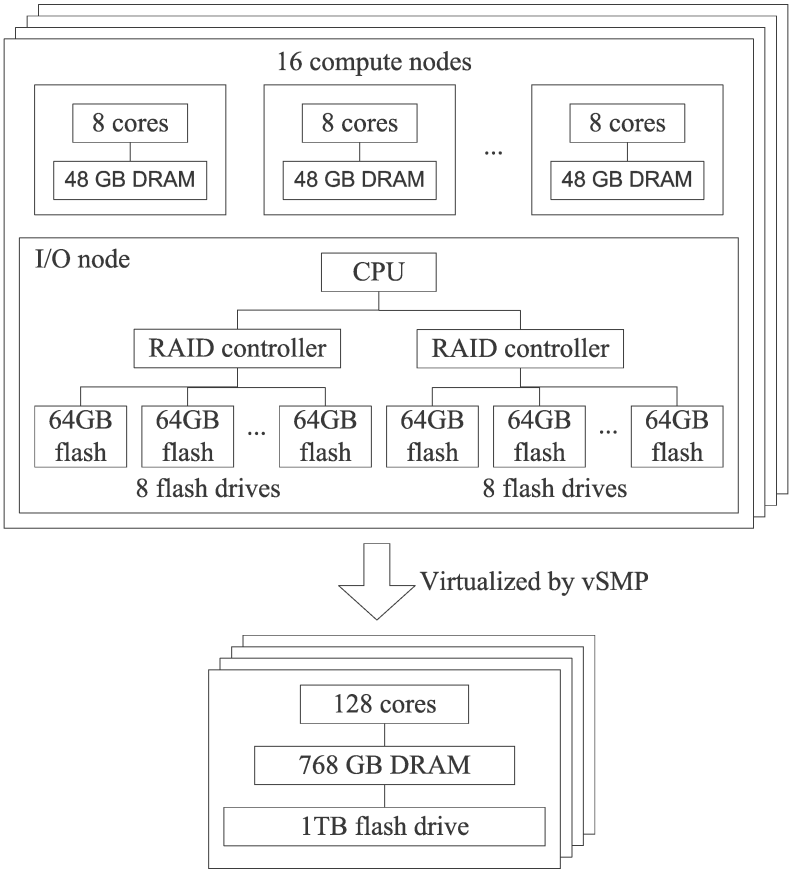

Figure 4.2: Physical and virtual structure of DASH supernodes. DASH has in total 4 supernodes IB interconnected of the type shown in the figure.

their data from spinning disks up to the shared memory in the memory hierarchy, a full 3 orders of magnitude closer to the CPU in terms of latency. Users with less than 1 TB of data can still avoid spinning disk and operate 2 orders of magnitude faster by loading their data on the flash of one supernode. And if a user uses the whole machine he can gain access of up to 7 TB of DRAM + Flash (3TB + 4TB) for truly large data analysis problems.

4.1.2 Cost efficiency

DASH is designed to provide cost-effective data-performance. The design focused the architecture on providing cost-efficient IOPS which should benefit all data-intensive applications. It is interesting to compare the three lowest levels of data hierarchy on DASH (the HDD, SSD, and virtually aggregated DRAM layer) to each other and some commercial offerings. Table 4.1 shows a cost efficiency comparison between DASH data hierarchy levels and two popular commercial products offered by 1) Fusion-I/O (ioDrive [49]) and 2) Sun Microsystems/Oracle (F5100 configuration-1 [43]).

Table 4.1: Cost efficiency comparison between DASH and commercial products.

	Generic HDD (SATA)	DASH-I/O node	DASH Super node	Fusion –IO	Sun – F5100
GB	2048	1024	768	160	480
MB/s/$	~0.4	0.16	0.49	0.12	0.07
$/GB	~0.15	19.43	112.63	41.06	90.62
IOPS/$	0.4-1.0	28	52	18	9
IOPS/GB	0.05-0.1	549	5853	725	828

The cost metrics in Table 4.1 are collated and averaged from different sources including the technical specifications of each product available from its vendor and reseller [49][43][139]. The listed prices of these products were observed on the first week of February, 2010. The second column (Generic HDD) was chosen to represent that category within a range of values (price, density, and speed varies

by vendor product). The cost of DASH I/O node includes the flash drives, the controllers, and the Nehalem processor; the cost of DASH supernode includes the cost of 16 dual socket Nehalem nodes, their associated memory, and the IB interconnect but not the I/O node (its performance was measured with RAM drive). The comparison to commercial products then gives an unfair cost disadvantage to DASH as the vendor's offerings are just storage subsystems and lack any substantial compute power - nevertheless, it is useful as a relative comparison. The third row (MB/s/$) can be seen as saying that bandwidth per dollar is more favorable for spinning disks and DRAM than for flash and DASH scores the best by this metric at all levels. The forth row ($/GB) says (common sense) that capacity per dollar is (in the order high to low) HDD (spinning disk), SSD (flash), DRAM and that DASH has the cheapest flash for the systems compared (the vendor system's don't have any general-use DRAM just some DRAM cache). The fifth row (IOPS/$) can be seen as saying that IOPS per dollar is more favorable for DRAM and flash than for spinning disk and DASH scores the best by this metric again. As shown on row two (GB) DASH also has more than twice as much flash capacity than either of the vendors. Row six (IOPS/GB) shows that because of having this more capacity the metric IOPS/GB looks better for the vendors at the flash but that is in part because they have less than 1/2 the flash (DASH still has the highest value in the row six category not due to flash but due to its virtual DRAM supernode layer). DASH then is a very high performing and cost-effective system compared to commercial offerings in the same space and since this chapter describes how to build and tune it from commodity parts, people in the market for such a data-intensive system could consider simply building their own DASH by this recipe.

4.1.3 Power efficiency

Power and cooling costs form a major part of large data center's operating cost. Power and cooling costs can even exceed the server hardware acquisition costs over the lifetime of a system. The power consumption of flash SSDs is low, making them the right choice for DASH. Table 4.2 compares power metrics between flash SSD, HDD, and DRAM.

Table 4.2: Comparison of power metrics between SSD and HDD.

	DRAM 7x2 GB Dimms (14 GB)	Flash SSD 64GB	HDD 2TB
Active Power	70 W	2.4 W	11 W
Idle Power	35 W	0.1 W	7 W
IOPS per Watt	307	712	35

The numbers in Table 4.2 were averaged from technical specifications of various products and independent hardware evaluation tests [139][65][140]. The second and the third rows are self-explanatory. The forth row compares the IOPS that can be performed per watt. Since drives are partly active and partly inactive during the course of an application's execution, one can say that in general the time savings resulting from flash come with an *additional* power savings over spinning disk. IOPS/Watt may be as much as two-orders-of magnitude better than spinning disk. The substantially higher IOPS of DRAM (an order of magnitude higher than flash) comes at a higher power cost. So if one wishes to optimize IOPS per Watt (or IOPS for operating cost) then a system like DASH may be considered.

Overall, it can be seen that the experimental system DASH is a powerful, high capacity and fast system design even by commercial standards, and offers cost-efficient, power-efficient IOPS for data intensive computing.

4.2 I/O system design and tuning

The DASH supernode (shared memory) results simply from deploying vSMP software on what is otherwise a standard IB connected system. This work mainly focuses on the design and tuning process for the I/O node describing how to choose the controller and tune the RAID system.

To evaluate the performance of storage systems, bandwidth and IOPS are both important metrics. Bandwidth measures sequential performance while IOPS

shows the throughput of random accesses. This section presents the whole tuning process of the DASH storage system. Since the target applications are characterized as intensive random accesses, this work biased towards achieving high IOPS more than bandwidth in the design. To pursue and measure the peak I/O performance of the system, RAID 0 is adopted for this work.

IOR [63] and XDD [141] are two of the most accurate, reliable, and well-known I/O benchmarks. Both are used to verify each other and their results were always similar in the tests. For each software and hardware configuration, four tests are run: sequential write, sequential read, random write and random read respectively.

Figure 4.3 summarizes a series performance results obtained relative to the starting baseline obtained by default settings, about 46K IOPS with 4KB blocks. Basic tunings led to 88K IOPS (1.9x of the baseline) random read rate with 4KB blocks out of one I/O node; this is only about 15% of the theoretical upper bound of 560K IOPS ($16x35K = 560K$ IOPS since the manufacturer spec is 35K IOPS random read per Intel ® X25-E SSD and each I/O node has 16 drives). It was figured out that a bottleneck came from the low-frequency processor embedded in the first RAID controllers (RS2BL080) and switched to simpler HBAs (9211-4i) and software RAID (using the fast Nehalem processor on each I/O node as the I/O controller rather than the embedded processor). This helped the system to scale linearly up to 8 drives, with obtained performance of about 255K IOPS (5.6x of the baseline). To keep the linear scaling up to 16 drives though one have to remove even the software RAID and handle the separated drives directly, which leads to (a little more than) theoretical upper-bound performance of 562K IOPS (12.4x of the baseline). The vSMP distributed shared memory system is able to exploit the shared memory of DASH as a single RAM drive and boost the performance again up to 4.5 million random read IOPS, ($98.8x$ of the baseline using DRAM in place of flash). Details of how these results were obtained are described in the following sections.

Since a single hard disk (HDD) can only do about 200 IOPS per disk (random read 4KB blocks) depending on manufacturer, it can be seen that DASH can

provide two orders of magnitude higher IOPS from its flash-equipped I/O nodes and yet another two orders of magnitude from aggregated DRAM as RAM disk. These options effectively fill the latency gap.

4.2.1 Single drive tuning

Before tuning the whole I/O system, let's start with tuning a single flash drive first. Table 4.3 shows some important tuning parameters for flash drives. It is also needed to tune the software components, such as I/O benchmarks and the operating system, for single-drive tests, which will be discussed later.

Write caching and read ahead on other system levels might not be helpful for an intensive random workload. However, the situation is a little bit different on the flash drive level. Since the internal structure of a flash drive is highly parallel and logically continuous, pages are usually striped over the flash memory array, prefetching multiple pages and background write-back can be very efficient, while disabling these options, especially write caching, could cause a dramatic performance drop [31].

Advanced Host Controller Interface (AHCI) is Intel ®'s API specification

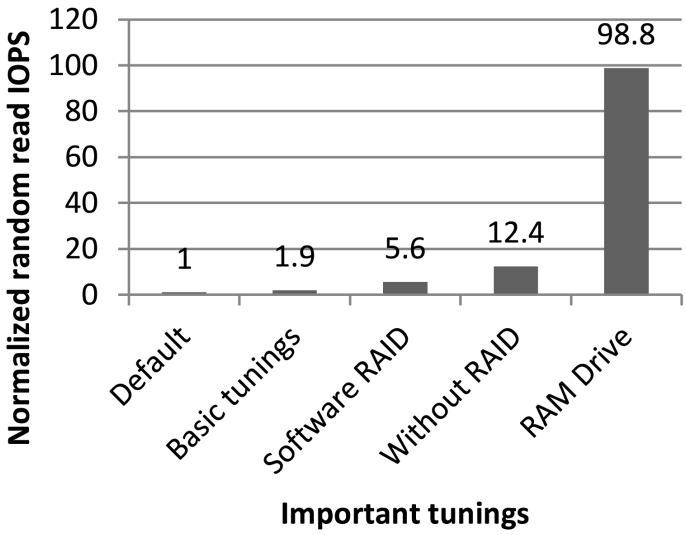

Figure 4.3: Random read performance improvements with important tunings.

Table 4.3: Important tuning parameters for flash drives.

Parameters	Descriptions	DASH setting
Write Caching	Write through or write back in the drive ram-cache	Write back
Read Ahead	Read the data into the drive ram-cache before they are requested according to the access pattern	On
AHCI	Advanced Host Controller Interface, API for SATA host bus adapters	On

Table 4.4: I/O test results of a single flash drive.

	Sequential Write (MB/s)	Sequential Read (MB/s)	Random Write (4KB IOPS)	Random Read (4KB IOPS)
Measured	203	261	10724	39756
Spec	170	250	3300	35000

for SATA host-controllers. One of its advantages is to enable Native Command Queuing (NCQ). In a traditional spinning disk, NCQ is designed to hold (and also schedule) the I/O requests not served by the disk fast enough. In flash drives, the purpose is the opposite. It is used to stock I/O requests in case the CPU is busy and cannot summit new requests in time [89]. For backward compatibility, AHCI is disabled by default in the system. Enabling the option led to more than 10x improvements on random read IOPS. Table 4.4 shows the I/O test results with a single flash drive. These performance numbers actually exceed the published specs of the Intel ® X25-E which are also listed in the table.

4.2.2 Basic RAID tuning

The tuning parameter space of the DASH storage system is large. To achieve maximum performance, one have to coordinate all the software and hardware components of the system: I/O benchmarks, the operating system, and hardware RAID. Table 4.5 summarizes the important tuning parameters of these components.

Usually the operating system will try to cache the data from/to disks for future uses. The RAID controller also has its own RAM cache for similar purposes. Unfortunately, cache doesn't always help. For example it may not help large-scale random I/Os (or even very large sequential I/Os) with low temporal locality. Even worse, it will introduce extra overhead on the data path. Direct I/O is enabled to bypass the OS buffer cache and the RAID cache is turned off.

There are quite a few APIs (libraries) one can use for I/O accesses. IOR supports four: POSIX, MPIIO, HDF5 and netCDF while XDD only supports POSIX. Since POSIX is the most common and typical in application code, it is chosen for the tests. MPIIO is also widely used in HPC community. Unfortunately, it doesn't support direct I/O.

Chunk size is decided according to the test type and the stripe size. Sequential tests are designed to measure the maximum bandwidth across all the underlying flash drives and the chunk size should be larger than the stripe size times the number of flash drives (16 in this case). In this work, 4MB is chosen, which is big enough for the stripe sizes. Random tests are designed to evaluate how well the system deals with small chunks of random access. Since the access unit (page size) of the flash drives is 4KB, it should be a reasonable (minimal) setting.

Queue depth also depends on the test type and the number of underlying flash drives. For sequential tests, since each request already covers all the underlying flash drives, it is set to 1 to guarantee a strict sequential access pattern. As for random tests, to maximize the throughput, 128 is chosen, which is large enough comparing with the number of flash drives (16 in this case), and hopefully can make a full use of each flash drive.

Table 4.5: Important tuning parameters for the DASH I/O system.

Components	Parameters	Descriptions	Final DASH setting
I/O Benchmarks	Cache Policy	Cached or direct I/O, use the OS buffer cache or not.	Direct I/O
	API	I/O APIs to access drives such as POSIX, MPIIO, HDF5 and netCDF.	POSIX
	Chunk Size	The data size of each request. I/O benchmarks usually generate fixed-sized requests.	4MB for sequential tests, 4KB for random tests
	Queue Depth	The number of outstanding I/O requests.	1 for sequential tests and 128 for random tests
Operating System	I/O Scheduler	Schedule and optimize I/O accesses. There are 4 algorithms in the 2.6 Linux kernel: CFQ (default), Deadline, Anticipatory, and No-op.	No-op
	Read Ahead	Read the data into cache before they are requested according to the previous access pattern.	Off
Hardware RAID	Cache Policy	Cached or direct I/O, use the RAID controller cache or not.	Direct I/O
	Write Policy	Write through or write back.	Write through
	Read Ahead	RAID-level read ahead.	Off
	Stripe Size	The block size in which RAID spread data out to drives.	64KB

There are 3 goals for I/O scheduler: merging adjacent requests together, reordering the requests to minimize seek cost (elevator scheduling), and controlling the priorities of requests. Since there is no drive head movement in flash drives, elevator scheduling is not necessary. Also, the system is not designed to run any

time critical applications and does not need prioritization either. In the experiments, the simplest No-op scheduler, which only proceeds request merging, always gave the best result.

One can set read ahead on 3 levels: operating system, RAID controller, and flash drive. The settings per SSD on the drive level was discussed above, but things are different on the other two levels. Read ahead is good for sequential performance, but it doesn't help random accesses. Sometimes it may even waste bandwidth with extra reads and hurt random performance. Since direct I/O was adopted and read ahead became irrelevant, it is turned off.

Again, it was already discussed about write-back and write-through on the drive level, but it is different on the RAID level. The common wisdom is that write-back is always better. However, it is only true for light workloads. With intensive random accesses, the write-back cache is not helpful. Also, the extra copy on the data path will hurt the performance. As a result, write-through is adopted on the RAID level.

To decide the stripe size is a difficult optimization. Usually, small stripe size will hurt sequential bandwidth because the start-up overhead dominates. For flash drives, it is even worse by causing serious fragmentation, which was proved to cause dramatic performance downgrading [31]. However, larger is not always better. After some threshold, large stripe size will limit the parallelism of I/O accesses and then the RAID system cannot exploit the bandwidth of all the underlying flash drives. Different sizes are tried from 8 KB to 1024 KB and it was found that 64 KB and 128 KB are the best configurations for this specific system and workload.

With the settings in Table 4.5, the system obtained the performance numbers for the stripe sizes of 64KB and 128KB shown in Table 4.6. As measured in Table 4.4, the random read performance of a single flash drive is 39,756 4KB IOPS. That means the upper bound for the whole IO node should be more than 600K IOPS, which is much higher than what has been obtained at this stage. The next sub-section will continue the adventure to figure out the problem.

Table 4.6: I/O test results with 2 different stripe sizes.

Stripe Size(KB)	Sequential Write (MB/s)	Sequential Read (MB/s)	Random Write (4KB IOPS)	Random Read (4KB IOPS)
64	1179	2199	3749	87563
128	1275	2056	3121	79639

4.2.3 Advanced tuning

As shown above, only about 15% of the maximum performance was achieved after all those tunings. What's the problem? It is suspected that the bottleneck might be the RAID controller. To implement the RAID function and other advanced features, also to reduce the CPU loads, the controller is embedded with a low-frequency processor (800MHz in this case). This small processor is enough for spinning disk, but not fast enough to work with flash drives. Are there any faster RAID controllers? To our best knowledge, it is the state-of-the-art RAID controller (Intel ® RS2BL080) one can get that is compatible with the drives. Another option is to use simple Host Bus Adapters (HBA) without embedded processors and share the power from the host CPU. The motherboard happens to have an on-board HBA similar to the RAID controllers but without embedded processor or hardware RAID function. Six flash drives were connected to compose a software RAID and $153,578$ 4KB IOPS was achieved, which is almost 2x of the hardware RAID performance. This confirmed the speculation.

The on-board HBA has a corresponding external version, which is rated higher than 150K 4KB IOPS by the vendor. Each HBA can connect 4 flash drives. The motherboard can hold 4 HBAs. By this means, with the same number (16) of flash drives, one can expect the random read performance of about 600K 4KB IOPS, which is very close to the upper bound.

With similar settings as the previous sub-section except replacing the hardware RAID with the HBAs plus the Linux software RAID, the tests were repeated. The random read performance scaled almost linear as expected at the beginning.

With 8 drives, about 250K IOPS were obtained, which is almost 3x as before. However, the scaling stopped after that. In Figure 4.4, we can see that there is a plateau from 8 to 16. Is it the RAID problem again? To answer the question, the software RAID were removed and the tests were performed directly on separate drives. This time the performance scaled almost linearly from 1 up to 16 drives. The highest performance was 562, 364 IOPS. Also removed the file system (XFS) and tested directly on the raw block devices. The results were almost the same. It seems the upper bound have been reached.

Table 4.7 lists all the I/O test results on 16 drives with and without RAID. You can see that the configuration without RAID is not only good for the random read test, but all the other tests. However, the performance with software RAID in fact is not too bad. By comparing with the results in Table 4.6, you will find that it still beats the original hardware RAID on almost all the tests. For the random tests, it achieved up to 5x the original performance. Though investigation of the RAID problem continues [52], it is safe to conclude that the software RAID configuration delivers a good balance between high performance and convenience.

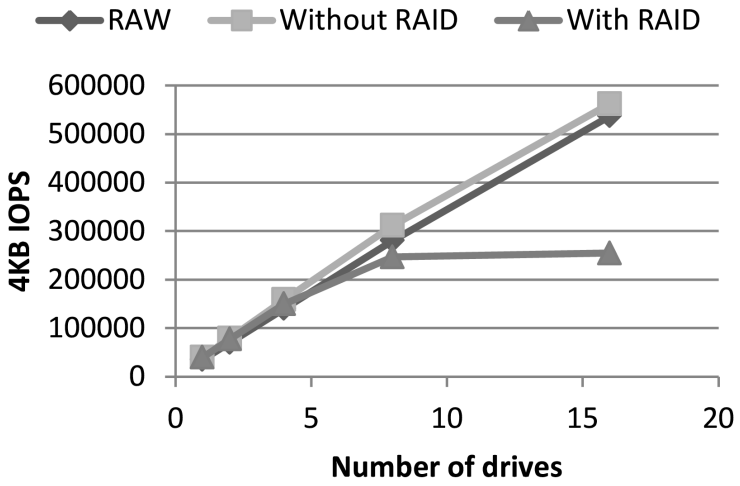

Figure 4.4: Random read performance with and without RAID. The configuration with RAID only scales up to 8 drives while the one without RAID can scale linearly up to 16 drives. Tests with raw block devices were also performed.

For the users who still need higher performance and don't care about the hassle to deal with 16 separate drives, the configuration without RAID is still an option and in fact there are some programming libraries around like STXXL [127] that can help to ease the job of managing the separate SSDs.

Table 4.7: I/O test results with and without RAID.

	Sequential Write (MB/s)	Sequential Read (MB/s)	Random Write (4KB IOPS)	Random Read (4KB IOPS)
With RAID	1395	2119	19784	254808
Without RAID	2958	3225	143649	562365

4.2.4 RAM drive

With the flash drives, the system obtained optimal results at the limit of the existing hardware technologies. However, with the special design of DASH, it is still possible for us to achieve even higher performance. As mentioned above, DASH adopts vSMP distributed shared memory software system to aggregate separate physical memories into a single virtual memory. Besides the part of the memory used by the vSMP software and reserved for cache, a user has access to about 650GB visible memories from any processor in each supernode. Such a big memory space can be used as a RAM drive by mounting with the RAMFS file system. Since DRAM accessed over IB is even faster than flash drives (by up to 3 orders of magnitude!), RAM drives are expected to achieve much higher performance and the results in Table 4.8 show this.

Table 4.8: I/O test results of the RAM drive.

Sequential Write (MB/s)	Sequential Read (MB/s)	Random Write (4KB IOPS)	Random Read (4KB IOPS)
11,264	42,139	2,719,635	4,495,592

4.3 Performance of real-world data-intensive applications

The behavior of I/O benchmarks as described above may be interesting and useful for comparison by simple metrics but the question remains "what is the implication for real applications"? To partially answer this question, one application core from predictive science and two full applications from data mining are investigated. This section presents the performance results of 1) external memory BFS, a common component in several predictive science graph-based applications 2) Palomar Transient Factory a database application used to discover time-variable phenomena in astronomy data. 3) Biological pathway analysis in an integrated data-mining of heterogeneous biological data framework. All three applications generate intensive random data accesses.

4.3.1 External memory BFS

Data in several domains such as chemistry, biology, neuroscience, linguistics, and social science, are implicitly graph structured or graphs may be induced upon them. For example, semantic tagged information is encoded as a graph where nodes represent concepts and labeled edges are relationships. Search engines model the World Wide Web as a graph, with web-pages as nodes and hyperlinks as edges. Researchers in linguistics use graphs to represent semantics expressed in sentences. Networks of roads, pipelines, neurons etc. can all be viewed as graphs. Moreover, due to technological advancements, scientists are increasingly harvesting massive graphs in their respective fields. For example, human interaction networks as

large as 400 million edges in size are already extant [44]. Information repository such as NIH's Neuroscience Information Framework (NIF) [90] is projected to have more than a billion edges. Web-graphs which are studied by social scientist, mathematicians, and linguistics can be on the order of tens of billions nodes. As semantic web gains prominence and natural language processing improves, we shall see an exponential growth in graph structured data sets.

A basic type of computation over graphs that appears frequently in all such domains is that of graph traversal. Although the nature and characteristics of a graph exploration varies across domains and even across problems within a domain, they are commonly modeled after breadth-first-search (BFS). Further, other domain specific problems such as finding complexes in protein-interaction network, clustering of web-graphs, computing distance-distribution in graph models etc. utilize BFS operation. Since the total size of the graph and the content associated with every nodes and edges can run up to the order of several tera-bytes, scalable and efficient BFS computation when graphs reside in external memory would help advanced research across all these domains. This problem in literature has been referred to as external memory BFS or EM-BFS.

The external memory package 0.39 implemented by Deepak Ajwani *et al.* [16] is adopted in the experiments. Table 4.9 shows the results of one of the algorithms, MR-BFS. A range of tests are executed on a dataset size of 200 GB and compared the performance of three different storage media (RAM drive, flash drives, and spinning disks) with similar and comparable configurations. The results showed that RAM drive is on average about $2.2x$ faster than flash drives, and flash drives are about $2.4x$ faster than spinning disks for an overall speedup of $5.2x$. The speedup is substantial but not as good as expected, which could be explained by the mix of bandwidth and latency bound (sparse and dense) accesses in traversing the test graph. As previous works [98] observed, write-intensive nature of an application might also be the cause.

Table 4.9: Average MR-BFS results on the Dash SuperNode from different storage media.

	RAM Drive	Flash Drives	Spinning Disks
Total I/O Time (sec)	854 (5.2x)	1862 (2.4x)	4444
Total Run Time (sec)	1917 (3.0x)	3130 (1.8x)	5752

4.3.2 Palomar Transient Factory

Astrophysics is transforming from a data-starved to a data-swamped discipline, fundamentally changing the nature of scientific inquiry and discovery. New technologies are enabling the detection, transmission, and storage of data of hitherto unimaginable quantity and quality across the electromagnetic, gravity and particle spectra. These data volumes are rapidly overtaking the cyber infrastructure resources required to make sense of the data within the current frameworks for analysis and study. Time-variable ("transient") phenomena, which in many cases are driving new observational efforts, add additional complexity and urgency to knowledge extraction: to maximize science returns, additional follow-up resources must be selectively brought to bear after transients are discovered while the events are still ongoing.

Current transient surveys like the Palomar Transient Factory (PTF) [111] and the La Silla Supernova Search [71] (100GB/night each) are paving the way for future surveys like the Large Synoptic Survey Telescope (LSST) [75] (15TB/night producing petabytes of data each year). The future sky surveys assess their effectiveness and scalability on current surveys such as PTF, in order to maximize the scientific potential of the next generation of astrophysics experiments. Two of the major bottlenecks currently confronting PTF are I/O issues related to image processing (convolution of a reference image with a new one followed by image subtraction) and performing large, random queries across multiple databases in order to best classify a newly discovered transient. PTF typically identifies on the order of 100 new transients every minute it is on-sky (along with 1000 spurious detec-

tions related to image artifacts, marginal subtractions, etc.). These objects must be vetted and preliminarily classified in order to assign the appropriate follow-up resources to them in less than 24 hours, if not in real-time. This often requires performing more than 100 queries every minute through 8 different and very large (˜100GB - 1 TB) databases. The response times of these queries are crucial for PTF. The forward query and the backward query are two most significant queries used repeatedly by PTF. The average times to run these queries on DASH and the existing production infrastructure used by PTF (with same cache-size, indexes) are provided in Table 4.10. The difference in query response times can be attributed to the random IOPS provided by SSDs which allow faster index scans of the database rather than sequential table scans. The two-order-of-magnitude improvement in response times makes it possible for PTF to keep up with real-time demands.

Table 4.10: Comparison of PTF Query response times on DASH and PTF production database with spinning disks.

Query type	Forward Query	Backward Query
DASH-IO (SDSC)	11ms (124x)	100s (78x)
Existing DB	1361ms	7785s

4.3.3 Biological pathways analysis

Systems level investigation of genomic information requires the development of truly integrated databases dealing with heterogeneous data, which can be queried for simple properties of genes as well as for complex biological-network level properties. *BiologicalNetworks* [17] is a Systems Biology software platform for analysis and visualization of biological pathways, gene regulation and protein interaction networks. This web-based software platform is equipped with filtering and visualization tools for high quality scientific presentation of pathway analysis results.

The *BiologicalNetworks* platform includes a general-purpose scalable ware-

house of biological information, which integrates over 20 curated and publicly contributed data sources including experimental data and PubMed data for eight representative genomes such as *S.cerevisiae* and *D.melanogaster*. *BiologicalNetworks* identifies relationships among genes, proteins, small molecules and other cellular objects. The software platform performs a large number of long-running and short queries to the database on postgres. These queries are a bottleneck for researchers on this domain when they have to work on the pathways using the visual interface. In the performance tests, some popular queries of *BiologicalNetworks* were run on three different media on SDSC DASH including hard disks, SSDs and memory (using vSMP).

Table 4.11: Query response times of popular queries in Biological Networks on different storage media (Hard disk, SSD and memory) and their speed-up in comparison to hard disk.

Query	Q2C	Q3D	Q5F	Q6G	Q7H
RAMFS (vSMP)	11338ms (1.42x)	62850ms (3.60x)	3ms (186x)	17957ms (1.54x)	211ms (5.64x)
SSD	11120ms (1.45x)	176873ms (1.28x)	11ms (50.73x)	24879ms (1.11x)	495ms (2.41s)
HDD	16090ms	226023ms	558ms	27661ms	1191ms

Again, as observed in the PTF queries (Table 4.10), the queries of the *BiologicalNetworks* also show improvement in their response times. However, speedup is not linear or constant across all the queries as each query uses a different query plan producing different quantity of results (or the number of rows scanned and selected from the relational database). Heavily random access patterns speedup by as much as two orders-of-magnitude while long sequential accesses run just a bit faster.

In summary, some real applications speed up between $5x$ and nearly $200x$ on DASH depending on the I/O access patterns and how much the application can benefit from the random IOPS offered by DASH.

4.4 More discussions on flash drives

4.4.1 Performance downgrading

Performance downgrading is one of the concerns about replacing spinning disks with flash drives. There are mainly two causes for the problem. First, fragmentation has proved to be very harmful to the performance [31]. Fortunately, with the high-end SLC flash drives, most of the performance downgrading is still acceptable, especially the random read performance. Furthermore, some test conditions in the above paper are extreme and not common in normal uses.

Also, filling up a new drive will also hurt the performance. A new drive out of factory might be marked as free. However, since there is no abstraction of free blocks in flash drives [3], the drive will be "full" permanently after each block is written at least once. This will keep the full cleaning pressure and downgrade the performance. To solve the problem, the operating system and the drive firmware have to support the TRIM instruction [134] to inform the drive when the content of a block is deleted. Linux has already supported this since the version 2.6.28. Intel ® already released a firmware update with TRIM for its similar product X25-M [61] and the result is promising [57]. Hopefully, the X25-E drives will be supported in a near future.

4.4.2 Reliability and lifetime

By system reliability, both functional failures and bit errors are concerned. Mean Time Between Failures (MTBF) is a widely-used metric for functional failure rate. Without movable mechanical parts, flash drives are more robust and easier to protect. The X25-E drives used in DASH have an MTBF of 2,000,000 hours [139]. As for bit errors, the raw Bit Error Rate (BER) of SLC NAND flash is about $10^{-9} - 10^{-11}$, commercial products usually apply Error Correction Code (ECC) with different strengths to lower the rate. The final error rate after ECC correction is called Uncorrectable Bit Error Rate (UBER) [84]. The UBER of X25-E is 10^{-15} [139]. That means you will get one bit flip in about 6 days if you keep reading with the sustained speed of 250 MB/s. For practical workloads, the time

will be much longer. Moreover, some products such as those from Fusion-IO or Pliant claim UBERs several orders of magnitude lower.

The lifetime of a flash drive is related to its reliability, especially BER. BER increases while a block ages because of writes, i.e. Program/Erase (P/E) cycles. After some point, the flash controller will disable the block. The typical expected lifetime for SLC is 100,000 P/E cycles [50]. Manufacturers usually apply wear-leveling to distribute writes evenly across all the blocks. Calculations indicate that under extreme use (constant write random access patterns at peak rate) the drives will not exhaust their write endurance for over 1 year. Real usage patterns will result in longer lives. To protect the system, people can adopt traditional methods such as RAID. Furthermore, flash lifetime can be predicted quite accurately with enhanced SMART (Self Monitoring, Analysis and Reporting Technology) including P/E cycle information.

4.4.3 Flash-oriented hardware and software

Flash-based SSD is a promising technology to replace traditional spinning disk. Its low latency and high throughput are going to improve the performance of storage systems dramatically. For example, in database systems, capacity is often traded for throughput. With flash drives' high throughput, it is possible to replace hundreds of small spinning disks with just a few large flash drives [3]. To release the full potential of flash drives, the related hardware and software, such as host peripheral chipset, interconnect, RAID, and operating system, have to be modified or even re-designed. Especially, it was found that RAID (hardware or software) is a limiting factor during the tuning process, and this work is not the first one to observe the phenomenon [3]. As referred to above, operating systems and drive firmware need to support TRIM instruction to avoid dramatic performance downgrading. With flash drives becoming widely accepted, these related technologies will be stimulated to improve soon.

4.5 Related work

4.5.1 ccNUMA machines

ccNUMA means Cache Coherent Non-Uniform Memory Access. It is a hybrid architecture combining the merits of SMP (Symmetric Multi-Processing) and cluster. With SMP, people can program in the same way as on their PCs. It is the most desired architecture for parallel programmers. However, such architecture is not scalable and usually limited by 32 processors/cores. To scale up, people usually group a bunch of SMP nodes together into a larger cluster. By this way, programmers might need to apply shared-memory programming model intra-node and message-passing model inter-nodes for optimal performance. ccNUMA machines try to turn the distributed memory on these SMP nodes into a single shared memory space by special hardware. There are a few commercial products around like the SGI Altix 4000 series, HP Superdome, and Bull NovaScale 5000 series [128]. With these machines, people can program across all the nodes in shared-memory model. However, these products usually adopt proprietary technology based on customized hardware, and need a long development period, which makes their ratios of performance to price pretty low. As discussed in the next sub-section, vSMP is a software implementation of ccNUMA and is much more cost efficient.

4.5.2 Distributed Shared Memory (DSM)

Since ccNUMA is an expensive solution, people try to achieve the same function with a software implementation called Distributed Shared Memory (DSM). The idea was first proposed and implemented in IVY [73]. During the late 1980s and early 1990s, there were a lot of projects, such as TreadMarks [6], Shrimp [19], and Linda [5], inspired by the idea and trying to improve in different ways. Though the idea is very attractive, these systems didn't get widely adopted. However, there appeared several commercial and academic DSM systems again recently [114][29][1]. It should be the right time to revisit the problem for several reasons. First, most of those old systems were developed in late 1980s and early

1990s and mainly worked with Ethernet. The high network latency limited their performance. With the low-latency inter-connect like Infiniband [60] today, the limitation is largely eliminated. Second, the workloads today are changing. Data intensive applications are becoming dominant, and the requirement for large shared memory is becoming stronger. Last but not least, most of the new systems exploit the virtual machine technology and implement the DSM layer under the operating system and right above the hardware. This might bring more opportunities to optimize. Also, it provides a single system image to the operating system and eases the management burden.

4.6 Conclusions and future works

We are entering the HPC era of data intensive applications. Existing supercomputers are not suitable for this kind of workloads. There is a 5-orders-of-magnitude gap in the current storage hierarchy. This work designed and built a new prototype system called DASH, exploiting flash drives and remote memory to fill the gap. Targeting at random workloads, this work tuned the system and achieved 560K 4KB IOPS with 16 flash drives and 4.5M 4KB IOPS with 650GB RAM drive. With 3 real applications from graph theory, biology, and astronomy, up to two-orders-of-magnitude speedup with RAM drives were attained compared with traditional spinning disks. As for cost efficiency, flash is cheaper than DRAM but more expensive than disk yet the cost of operation (power) of flash is less than spinning disk.

DASH is a prototype system of the even larger machine called Gordon, which has much more flash drives and memory. To achieve good performance with such a huge system, it is needed to figure out how to scale up the storage system and the DSM system.

New storage media like flash and PCRAM is a hot research direction. How to integrate flash into the storage hierarchy is one of the difficult topics. It can be used as disk replacement, memory extension, disk cache, and more. It will be an interesting topic to investigate what is the best way to use flash in HPC systems.

This chapter, in part, is a reprint of the material as it appears in the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10), a joint work with Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett, and Allan Snavely. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Performance Characterization of Flash Storage System

As shown in the previous chapter, flash drives are promising for data intensive HPC applications. However, what is the best way to integrate flash technology into the existing HPC architecture is still an open problem. Since most the existing hardware and software were designed for slow spinning disks, there will be a lot of surrounding components like operating system and RAID (hardware or software), which might not be able to catch up with the high-speed flash drives.

To explore the potentials and issues of the flash technology, A large parameter space was swept by fast and reliable measurements to investigate different design options. Also, some lessons learned and suggestions for future architecture design are presented.

5.1 DASH System Architecture

DASH is composed of four so-called supernodes connected by InfiniBand [60]. Each supernode is a 16-way cluster with 16 compute nodes and 1 IO node. All the nodes are equipped with two Intel ® Nehalem quad-core 2.4GHz CPUs and 48GB DDR3 memory. Each IO node has in addition 16 Intel ® X25-E [139] 64GB flash-based SSDs (Solid State Disks), with the total capacity of 1TB, serving the affiliated supernode. The whole supernode (including the compute nodes and the

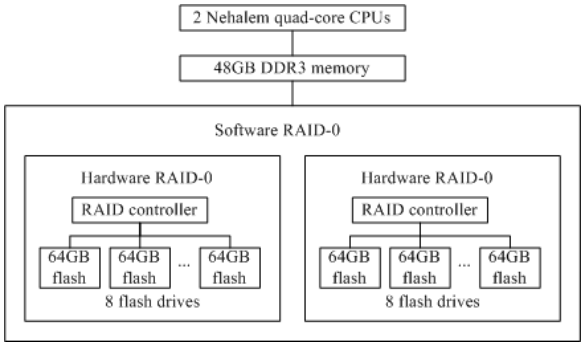

Figure 5.1: The original design of IO nodes. Each eight drives are grouped by a hardware RAID controller into a hardware RAID-0. Another software RAID-0 is set up on top of the two hardware RAIDs. The best random read IOPS achieved is about 88K, which is about only 15% of the theoretical upper bound.

I/O node) is virtualized into a single system image by the vSMP system from ScaleMP ® inc. [114]. From users' perspective, a supernode has 128 cores, 768GB main memory, and 1TB flash drives. A single Linux image runs on top of the vSMP system. In fact, DASH is just a prototype system for the even larger Teragrid system called Gordon coming next year. Gordon will have more (32) and larger (32-way) supernodes.

DASH is the prototype solution to the memory-disk latency gap problem. Two innovations are adopted to fill the gap: flash drives and distributed shared memory. Flash drives can accelerate IO by about 2 orders of magnitude in term of latency. Access to remote DRAM memory provided by the vSMP system can improve the performance by another order of magnitude albeit to a smaller(768GB) pool of storage. This work will focus on the flash technology and try to explore the design space.

5.2 Flash-based IO Design Space Exploration

Figure 5.1 shows the original design for IO nodes using hardware RAID plus software RAID. To connect the 16 flash drives, two Intel ® RS2BL080 PCIe 2.0 RAID controllers are adopted with 8 up-to-6Gb/s SAS/SATA ports each. Every

eight drives are configured as a hardware RAID-0. Another software RAID-0 was set up above the two hardware ones. There are several RAID levels like 0, 1, 5, 6. Since the flash drives will be used as a working area instead of a permanent storage system, redundancy or reliability is not guaranteed. To pursue high performance without any parity computation overhead, this work only investigates RAID-0. Without any tuning, the out-of-box random read performance is about 46K IOPS with 4KB requests. Exhaustive tuning led to about 88K 4KB IOPS, which was about 2x improvement. However, since each X25-E drive can perform about 35K 4KB IOPS, the upper bound performance of 16 drives should be about 600K and only about 15% of the upper bound was obtained, which was quite disappointing. After some investigations, it is suspected that the bottleneck could be the embedded 800MHz IO processor of the RAID controller, which was designed for spinning disks and might not be able to work with the much faster flash drives. One obvious solution to the issue is to use faster RAID controllers. However, RS2BL080 was already the best RAID controller one could find at the time the machine was built (fall 2009). It seems the existing hardware RAID controllers are not ready for flash drives yet. In fact, this work is not the only one encountering the issue. Previous work [3] observed the same problem. Another workaround is to avoid hardware RAID and adopt software RAID instead. By this approach, one can leverage the powerful Nehalem processors of the host. To verify the hypothesis that this might improve matters, 6 flash drives were attached to the on-board HBA (Host Bus Adapter) of the motherboard and repeated the test with software RAID. Only 6 drives could easily achieve about 150K IOPS, which is about 2x of hardware RAID performance.

With the hypothesis confirmed, the controllers were replaced by simple HBAs (LSI ® 9211-4i) with 4 up-to-6Gb/s SAS/SATA ports each. Besides the software/hardware RAID issue, it is believed there are more design dimensions, such as stripe size, stripe width (number of drives), file system, IO schedulers, queue depth, write caching, and read ahead, that are critical to the performance of the flash storage system. To explore the complete parameter space might cause exponential explosion in design space and seems infeasible. In fact, it is possible

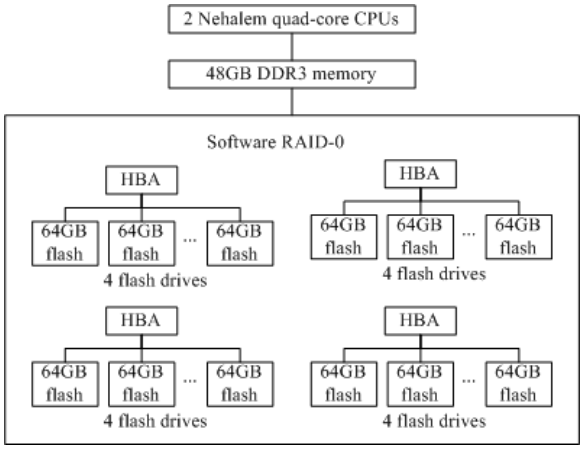

Figure 5.2: The IO node design after switching to simple HBAs. All 16 drives are set up as a single software RAID-0. The random read IOPS was improved by about 4x comparing with the original design up to about 350K.

to reason about the appropriate range for each dimension and limit the space to explore. The following subsections will discuss how to choose the parameter space, set up the experiments, and what the results imply.

5.2.1 Experiment Configurations

Figure 5.2 shows the IO node configuration after switching to the LSI ® HBAs. Instead of the hierarchical structure with hardware RAID plus software RAID, this time simply group up all 16 drives with a single software RAID-0. There are quite a few IO benchmarks around and one of the most accurate and stable benchmarks XDD [141] was chosen. To measure the pure performance of the flash drives without interference from OS buffer cache, direct IO is adopted across all the tests.

Table 5.1 shows all the parameter dimensions measured and their tested values. With direct IO, write caching and read ahead become irrelevant. Since targeting applications with large amount of small random accesses, write caching and read ahead are not helping.

Stripe size is the chunk size in which a RAID spreads out data into different drives. The range from 16KB to 256KB should cover the reasonable sizes. Stripe

Table 5.1: Parameter Dimensions and Their Values

Parameters	Descriptions	Values
Stripe size	The chunk size of RAID	16KB, 64KB, 256KB
Stripe width	Number of drives	1, 2, 4, 8, 16
File system	With or without XFS	Raw, XFS
Read ahead	Linux prefetching	Off, On
IO scheduler	Linux IO scheduling algorithms	No-op, CFQ, Deadline and Anticipatory
Request size	Size of IO requests	4MB for sequential tests; 4KB for random tests
Seek position	Start addresses of IO requests	sequential-seeking, random-seeking
Queue depth	Number of outstanding IO requests	1, 4, 16 for sequential tests; 32, 128, 512 for random tests
IO operation	Read or write	100% read, 100% write

width is the number of drives in a RAID. This work tested the numbers from 1 to 16 to investigate the performance scalability.

File systems may add some overhead to the system. It will be interesting to compare the situation with and without file system to see how big the overhead is. XFS [142] is a file system designed for large compute and storage system with excellent performance and scalability. Read ahead is the mechanism the operating system tries to prefetch data the user will request. As referred to above, read ahead should not be effective with direct IO, which is confirmed by measured results (not shown in this chapter). Linux has four IO scheduler choices: No-op, CFQ, Deadline and Anticipatory. This work explored to see how these scheduling algorithms perform with flash drives.

Request size means how big the requests generated by the IO benchmark (XDD in this case) are. 4MB should be large enough to guarantee the sequential access behavior and 4KB is a common size used in random tests and is found in many applications. Seek positions are the start addresses of the IO requests; these can be sequential or random. One thing worth mentioning is that random seeking does not necessarily mean random accesses. Random-seeking with 4MB requests still generates sequential accesses i.e. the pattern is occasional repositioning of

start address followed by a long sequential access. Queue depth is the number of outstanding IO requests that can be outstanding at a given time. Small numbers like 1, 4, 16 should be enough for sequential tests. Random tests will try large numbers like 32, 128, 512 to see what the performance impact is. The final dimension is read or write. This work only considers 100% read and 100% write. 14 combinations of the above four parameters are tested as follows. The following sections will refer to them as test types and use the numbers from 1 to 14 to present them in some of the figures.

- 4MB sequential tests with sequential seeking, queue depth 1, read and write;
- 4MB sequential tests with random seeking, queue depth 1, 4, and 16, read and write;
- 4KB random tests with random seeking, queue depth 32, 128, and 512, read and write.

Each test is repeated five times. There were in total 16,800 runs (it can be seen why some search-space limiting is needed). By limiting each run to 10 seconds, it is feasible to finish the entire data gathering experiment in about four days - analysis took a lot longer.

For each run, the benchmark will report both bandwidth and IOPS (Input/Output Per Second). Though people usually talk about bandwidth of sequential tests and IOPS of random tests, they are in fact the same thing and can be translated to each other as long as you know the request size. Since the sequential tests and random tests are using different request sizes, the bandwidth numbers will be preseted for a uniform view while talking about both sequential and random tests in the following sections.

5.2.2 Data Pre-processing

As referred to above, each test is repeated five times. After collecting the data, the data was pre-processed and outliers were searched for with Chauvenet's criterion [30]. According to the criterion, with five measured data, one can be

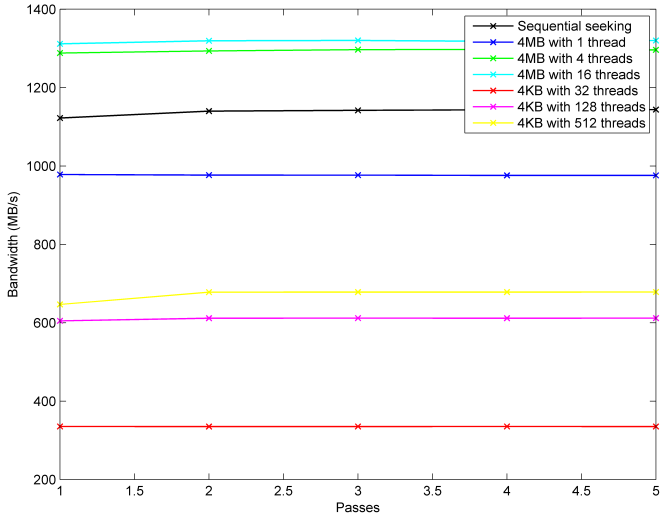

Figure 5.3: Average bandwidth over five passes of read tests. The first pass tends to off the trend.

classified as an outlier if it is 1.65 standard deviations away from the mean value. With the criterion, 1155 outliers were found out of the 16,800 measured data (about 7 percent).

Figure 5.3 and Figure 5.4 show the average measured values over the 5 passes for read and write tests respectively. Here we can observe some interesting phenomena. The first value of each test tends to be off the trend. It seems the performances of the flash drives are quite sensitive to the pre-conditions, but they can adapt to the conditions very fast right after the first run. The write performance is more sensitive than the read one, which is almost constant.

With the above observations, all the data of the first pass were dropped and the Chauvenet test was repeated. This time no outlier could be found at all. Figure 5.5 and Figure 5.6 show the coefficients of variation across all the tests before and after the change. The data became much more stable after removing the outliers.

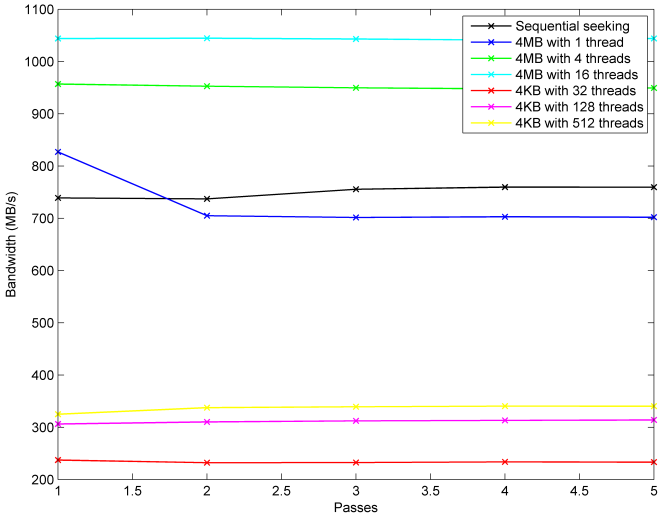

Figure 5.4: Average bandwidth over five passes of write tests. The first pass tends to off the trend. The write tests are more sensitive to pre-conditions but can adapt quickly right after the first run.

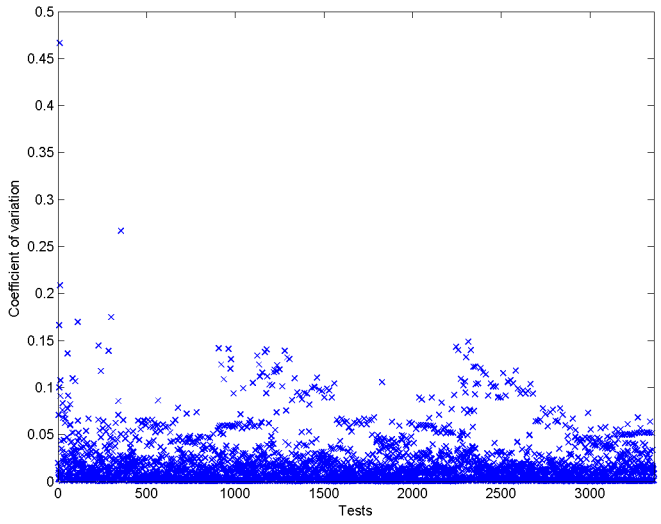

Figure 5.5: Coefficients of variation before outliers removal. With Chauvenet's criterion [30], 1155 outliers were found out the 16,800 measured data.

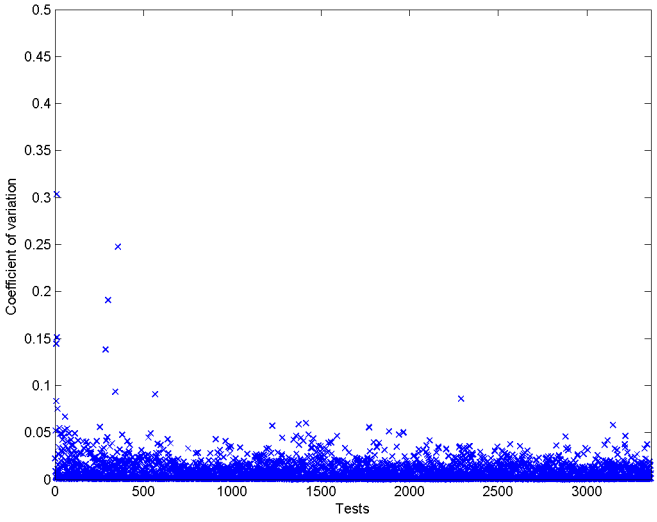

Figure 5.6: Coefficients of variation after outliers removal. After removing the first pass of each test, all the outliers are removed.

5.2.3 Stripe Size

The stripe size of a RAID is critical for performance. Small sizes are easier for applications to make use of the parallelism across the composing drives and increases the resultant bandwidth. However, too small sizes may cause significant overhead of striping and IO processing.

Figure 5.7 shows the average bandwidth with different stripe sizes. The X axis represents the test types referred above. We can see that 16KB is too small to achieve reasonable performance. The performance of 64KB and 256KB are almost the same. 256KB is a little bit better.

Since the random test size (4KB) is smaller than all the stripe sizes, a single request would not be striped across more than one drive. However, the queue depths are all larger than the drive number (16) and the requests should spread out evenly onto different drives. In this situation, smaller stripe sizes cannot take advantage of parallelism. Instead, they suffer from the striping overhead. For sequential tests with the request size of 4MB, all the stripe sizes tested can benefit from IO parallelism. Within the range, smaller stripe sizes will suffer from the striping overhead again. As a conclusion, the optimal stripe size depends on the

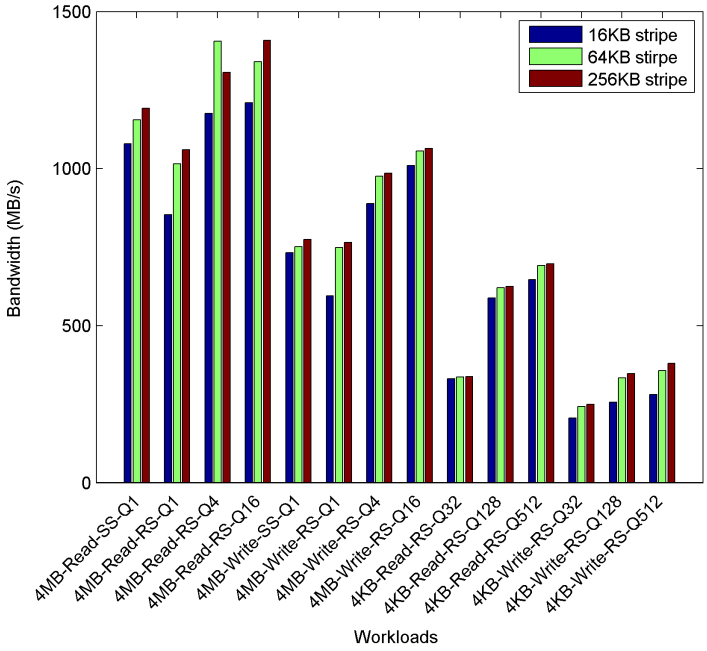

Figure 5.7: Average bandwidth with different stripe sizes. Deciding stripe size is a trade-off between parallelism and striping overhead.

application characteristics. The sweet spot may be the largest stripe size to make the requests span over all the drives. Of course the queue depth is also a factor. More discussion of this will be in the queue depth section.

5.2.4 Stripe Widths and Performance Scalability

The stripe width (the number of composing drives) of a RAID is another important factor for performance. Ideally, one would like to see the bandwidth or throughput (like IOPS) scale up with the number of drives. This work is especially interested in the IOPS performance of the random tests for a couple of reasons. First, the target applications are dominated by small random accesses. Also, since the IOPS number of a single flash drive is already pretty high compared to spinning disks, scaling the performance up can be challenging for the surrounding components such as RAID. In this section, we will see how the random IOPS scale up

from 1 drive to 16 drives.

Figure 5.8 shows the random read IOPS scaling over the number of drives. The good news is that comparing with the 88K IOPS result of the original design with hardware RAID, the tuning managed to improve the performance by about 4x, leveraging the horsepower of the Nehalem host processors. How about the scalability? Here things become more tricky. The performance scales almost linearly up to 8 drives, which is great. However, it almost stops scaling after 8 drives. With 16 drives, the performance can only scale up a little bit to about 350K IOPS from 250K with 8 drives.

Further investigations showed that the issue origins from the high IOPS of flash drives, which introduces a flood of hardware interrupts. The existing Linux kernel and HBA drivers do not work well enough to balance the workload to all the CPU cores. To be more specific, there is an existing issue [24] in the current kernel to configure interrupt binding for devices using message signaled interrupts (MSI) [86] but without MSI per-vector masking capability, which is the case for the HBAs. To work around the issue, one can disable the MSI and fall back to the legacy pin-based interrupt. Figure 5.9 shows the same results with the new configuration. With MSI disabled, now the random read performance can scale almost linearly up to 16 drives with the queue depth of 512. The peak performance is about 460K IOPS, which is about 80% of the theoretical upper bound (600K IOPS). However, the legacy interrupt may have some disadvantages such as limited number of interrupts and higher overheads. As you might have noticed, the performances with queue depth of 32 drop about 2x. That is because there are not enough outstanding requests to overlap and hide the long overheads in these tests.

The curves with the same queue depth are represented with the same color in both the figures. It is easy to see that the queue depth is very important. More threads usually means better performance.

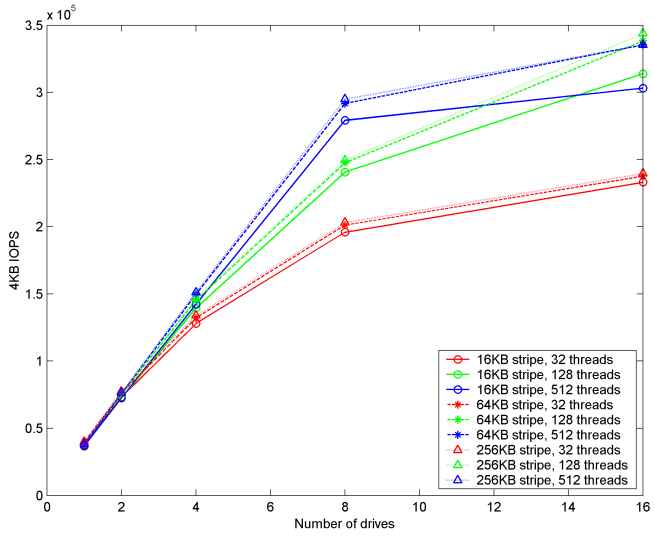

Figure 5.8: Random read IOPS scaling over drive amount. It scales almost linearly up to 8 drives but not after that.

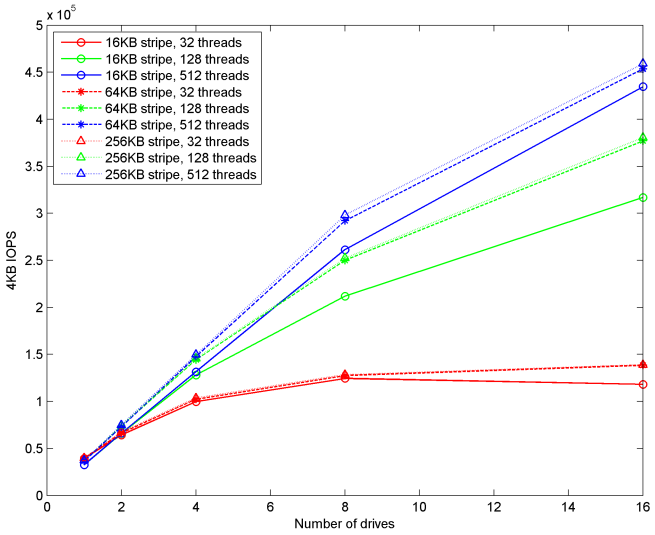

Figure 5.9: Random read IOPS scaling over drive amount without MSI-X. With MSI disabled, it scales almost linearly up to 16 drives.

5.2.5 File Systems

File system is an important factor at the operating system level. Since most of the existing file systems were designed before flash drives came into batch production, they might not be able to work with flash drives well. It will be interesting to figure out what the overhead of existing file systems will be. This work uses XFS [142] as an example. XFS was designed for large compute and storage system from day one and is famous for its excellent performance and scalability. It will be a good fit for the large-scale data-intensive supercomputer if its overhead on flash drives is small enough.

Figure 5.10 compares the bandwidth with and without XFS. The sequential performances (the first eight categories in the figure) are almost the same. That is because the overhead of the file system is amortized because of the large request size. However, the overhead becomes dominant in the random tests and XFS performs quite poor. One interesting phenomenon is that the random write performances (the categories 10, 12, and 14 in the figure) drops dramatically, whose reason is still under investigation. It will be interesting to also compare the performance with other file systems such as ext4 [42] or Brtfs [23]. It is one of the future topics.

5.2.6 IO Schedulers

For traditional spinning disks, people developed all kinds of IO scheduling algorithms mainly to serve three purposes: adjacent request merging, request reordering (elevator scheduling), and request prioritizing. Most of these optimizations are redundant for flash drives. For example, since there is no head moving and the seeking cost is trivial for flash drives, elevator scheduling is not necessary. Request prioritizing is not needed either by the target applications. Without any benefits, these optimizations might even introduce unnecessary overhead for flash drives. There are four IO schedulers in Linux: No-op, CFQ (Completely Fair Queuing), Deadline, and Anticipatory. No-op is the simplest one with only request merging function. This work tries to figure out if the other advanced algorithms can bring any benefits for flash drives.

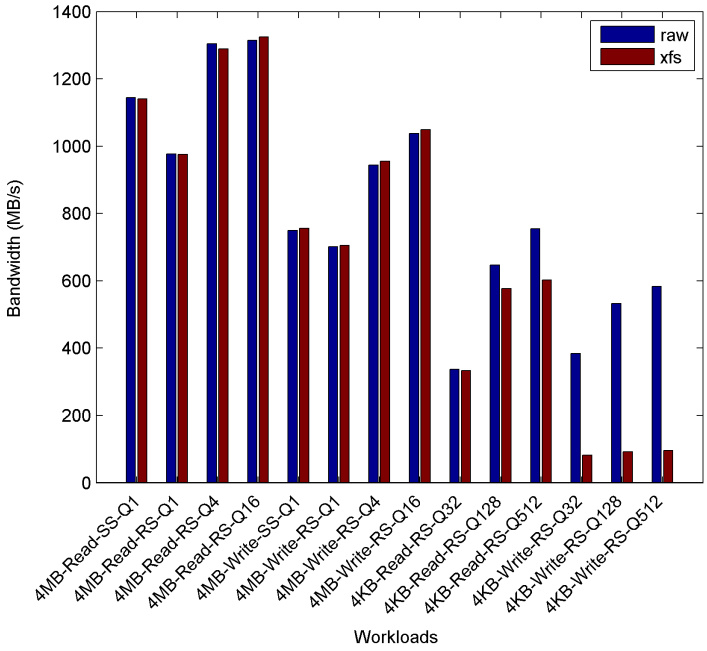

Figure 5.10: Average bandwidth with and without file system. The sequential performances with and without XFS are almost the same while the XFS's random (especially write) performances are worse.

Figure 5.11 shows the average bandwidth with different IO schedulers. Deadline is the best for sequential (random-seeking with 4MB requests) tests; No-op is the best for random (with 4KB requests) tests; CFQ varies a lot; while Anticipatory is systematically bad. We can see that advanced optimizations designed for spinning disks are not necessary for flash drives. Simple algorithms like No-op and Deadline work best.

5.2.7 Queue Depths

A traditional spinning disk is a serial device, which can only access one data block at a given time. Flash drives are totally different. Even a single drive may contain tens of data buses and packages internally, which can be accessed in parallel. To explore the full potential of a flash drive, a user has to utilize

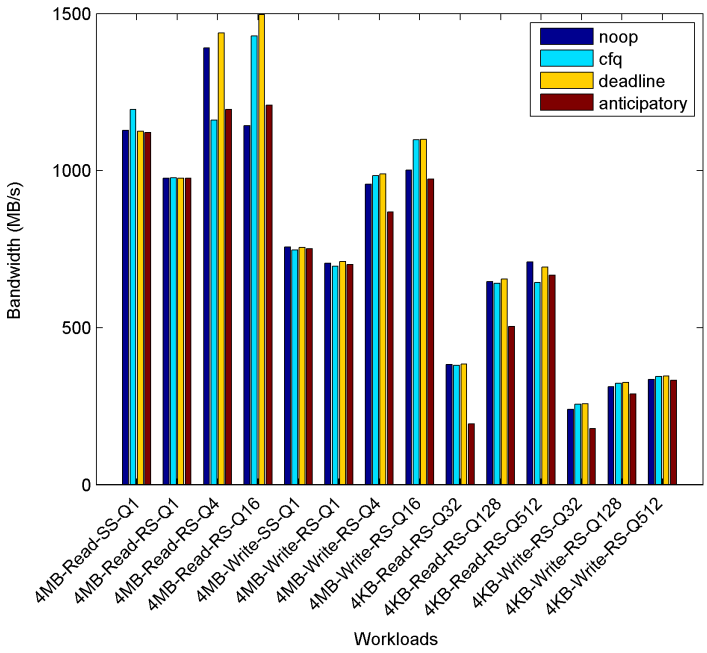

Figure 5.11: Average bandwidth with different IO schedulers. Simple algorithms like No-op and Deadline work best. Most advanced optimizations designed for spinning disks, such as elevator scheduling, are not necessary, even harmful for flash drives.

asynchronous IO or multi-thread to keep the drive busy.

Figure 5.12 and Figure 5.13 show the average bandwidth with different queue depths for sequential and random tests respectively. Both figures are logarithmic. All the tests (sequential or random, read or write) shared a similar rising trend with increasing queue depth. Even though the request size (4MB) of the sequential tests is large enough to span all the composing drives, higher queue depth can still improve bandwidth. The extra performance improvement comes from the internal parallelism of each drive. Random tests have a similar trend until the queue depth reaches 128. This makes sense because 512 is a big number for 16 drives with tens of internal packages each. In another word, queue depth of 512 exceeds the aggregated parallelism of the 16 drives.

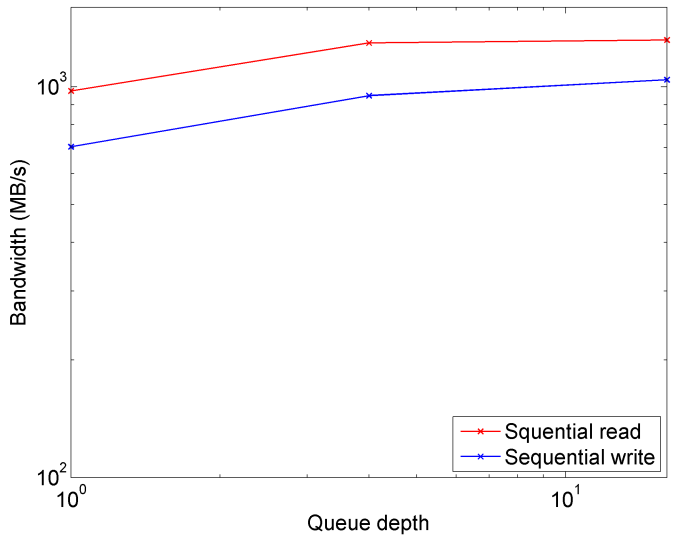

Figure 5.12: Average bandwidth of sequential tests with different queue depths 1, 4 and 16. Although the request size (4MB) can span across all the drives, higher queue depth can still improve bandwidth because of the internal parallelism of each drive.

5.3 Conclusions

Data-intensive HPC applications are becoming more and more common. Existing memory storage hierarchy has a 5-order-of-magnitude gap between memory and spinning disks, and is not able to respond to the challenge well. Distributed shared memory and high-speed flash drives are adopted to fill the gap, and the prototype system called DASH was built, which is a Teragrid resource. This work focused on the flash technologies and tried to explore the complete design space of a flash storage system. A large multi-dimensional parameter space was swept to figure out how the other system components, such as operating system and RAID (hardware and software), interfere with flash drives. These exhaustive searching managed to improve the performance from the original design by about 9x. However, it is found that some existing technologies like RAID don't fit the new technology very well. Future work includes to keep investigating the origins of these limitation and try to remove them and release the potential of the flash

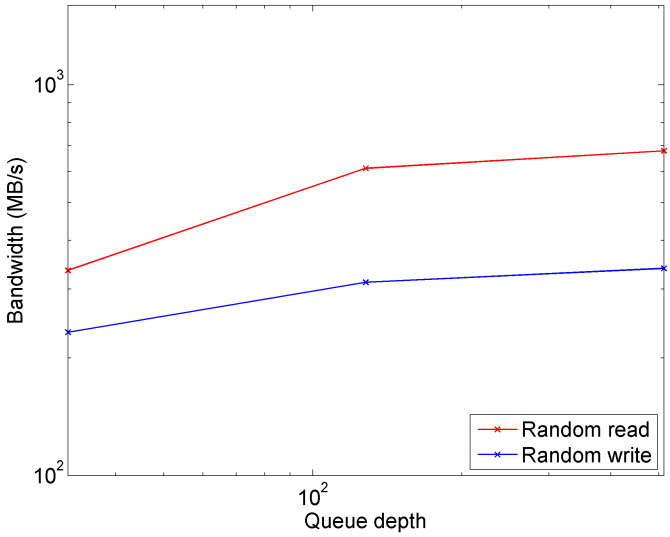

Figure 5.13: Average bandwidth of random tests with different queue depths 32, 128 and 512. Performance increases until the queue depth 128 only because 512 exceeds the aggregated parallelism of the 16 drives.

technology. Other emerging storage technologies such as PCM (Phase Change Memory) and STTM (Spin-Torque Transfer Memory) are also interesting. The goal is to re-shape the memory storage hierarchy to fit the high performance data trend ahead.

This chapter, in part, is a reprint of the material as it appears in the 2010 Teragrid Conference (TeraGrid'10), a joint work with Jeffrey Bennett and Allan Snavely. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Performance Prediction of HPC Applications on Flash Storage System

As the gap between the speed of computing elements and the disk subsystem widens, it becomes increasingly important to understand and model disk I/O. While the speed of computational resources continues to grow, potentially scaling to multiple peta-flops and millions of cores, the growth in the performance of I/O systems lags well behind. In this context, data-intensive applications that run on current and future systems depend on the ability of the I/O system to move data to the main memories. As a result, the I/O system becomes a bottleneck for application performance. Additionally, due to the higher risk of component failure that results from larger scales, the frequency of application check-pointing is expected to grow and put an additional burden on the disk I/O system [95].

Emergence of new technologies such as flash-based Solid State Drives (SSDs) presents an opportunity to narrow the gap between speed of computing and I/O systems. With this in mind, San Diego Supercomputer Center (SDSC) is investigating the use of flash drives in a new prototype system called DASH [92][53][52]. This chapter applies and extends the PMaC prediction framework to model disk I/O time on DASH. The methodology consists of the following three steps: 1) attain an application's I/O characteristics on a reference system; 2) using a config-

urable I/O benchmark, collect statistics on the reference and target systems about the I/O operations that are relevant to the application; 3) calculate a ratio between the measured I/O performance of the application on the reference system with respect to target systems to predict the application's I/O time on thetarget systems without actually running the application on the target system. This **cross-platform** prediction can greatly reduce the effort needed to characterize the I/O performance of an application across a wide set of machines and can be used to predict the I/O performance of the application on systems that have not been built yet. The cornerstone of this approach is that the I/O operations in the application have to be measured once on the reference system. The target systems then need only to be characterized by how well they can perform certain fundamental I/O operations, from which the I/O performance of the application on the target system can be predicted.

A data-intensive application benchmark called MADbench2 [21] is used as a case study here. The proposed methodology is evaluated by predicting the total I/O time of MADbench2. An I/O benchmark called IOR is used to characterize I/O operations of MADbench2 on target systems. The results show the methodology has prediction errors in the range of 8.59% to 20.66%, and the prediction error for total I/O time is 14.79%. The rest of the chapter is organized as follows: Section 6.1 gives an overview of the I/O performance prediction methodology; Section 6.2 describes the workloads and systems used for evaluation of the methodology and Section 6.3 presents the results of the evaluation; Section 6.5 presents conclusions and future work and finally, Section 6.4 presents related work.

6.1 Methodology

Given an application, a reference system, and target systems for which prediction is required, Figure 6.1 shows the modeling and prediction work flow used in the experiments. As shown in this figure, using PEBIL [72], a binary instrumentation tool developed in PMaC, one should first instrument all I/O calls in the application. The instrumented application is then executed on the reference

system and the application's I/O profile is stored for further analysis. The profile contains the time spent in all I/O calls. Additionally, one should also collect data that pertains to each call. For example, data size for read/write calls are collected. For seeks, the seek distances will be collected. Next, each I/O call is simulated using I/O micro benchmarks such as IOR [115]. The time spent by the I/O micro benchmark on the reference system and target systems are collected. An I/O ratio is calculated as shown in Equation 6.1. This ratio is the prediction for the predicted speedup or slowdown of the application's I/O on the target system relative to the reference system. One then use these ratios, as shown in Equation 6.2, to predict the applications total I/O time on target systems. To calculate accuracy of the predictions, one can run the application on the target systems and compare the

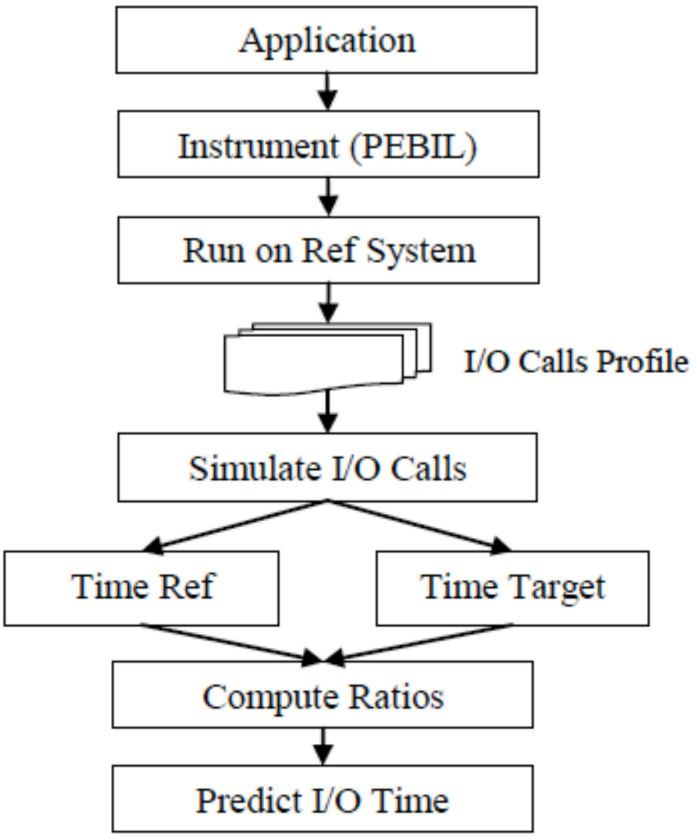

Figure 6.1: Methodology Overview.

predicted times with the actual time spent in I/O. This work extends the PMaC prediction framework to model patterns of I/O operations and predict I/O time on flash storage systems.

For each I/O call i,target system x, calculate ratios as follows:

$$Ratio_{i,x} = \frac{IOTime_{i,x}}{IOTime_{i,reference}} \tag{6.1}$$

For each target system x, calculate predicted total time spent in I/O as follows:

$$PredictedTime_x = \sum_{i=0}^{n} Ratio_{i,x} * ApplicationTime_{i,reference} \tag{6.2}$$

6.2 Experimental Workload and Systems

6.2.1 Workload

MADbench2 is a benchmark that is derived from Microwave Anisotropy Dataset Computational Analysis Package (MADCAP) Cosmic Microwave Background(CMB) power spectrum estimation code [20]. CMB data is a snapshot of the universe 400,000 years after the big bang. MADbench2 tries to simulate the most computationally challenging aspect of MADCAP to calculate the spectra from the sky map. To this end, MADbench2 retains the full complexity of computation, communication, and I/O, while removing the redundant details of MADCAP. The nature of the large calculations required for CMB data means that the large matrices used do not fit in memory. As a result, the benchmark uses an out-of-core algorithm. Each processor requires enough memory to fit five matrices at a given time. MADbench2 stores the matrices to disk when they are first calculated and reads them back into memory when required.

In this work, MADbench2 was configured to run in synchronous I/O mode with concurrent readers/writers. The application is configured to run with 25 MPI tasks using POSIX and MPIIO APIs in separate runs. A total of 8 matrices of 7.2 GB size are used and stored in a single shared 57.6 GB file. These 8 matrices are

distributed among the 25 MPI processes, and thus, each process works on 2.3 GB of data. The application makes I/O calls in three distinct phases. During phase 1, the matrices are written to disk. During phase 2, the matrices are read back and updated contents are written back to disk. Finally in the last phase, the matrices are read back from disk. Thus, a total of 16 reads/writes of size 288 MB are issued by each process.

6.2.2 Systems

DASH with Flash – Target System In the DASH system, flash drives are attached to the batch nodes and the I/O nodes. Each batch node is equipped with two Nehalem 2.4GHz quad-core processors and 48GB DDR3 memory. One Intel ® X25-E 64GB flash drives is attached directly to a batch node. As for the I/O nodes, they have the same processors and memories as the batch nodes but more flash drives. Specifically, 16 Intel ® X25-E 64GB, totally 1TB, flash drives are installed on each I/O node and set up as a software RAID-0. The XFS file system is applied to the flash dives on DASH.

Jade – Reference System Jade is a Cray XT4 system and has a total of 2152 compute nodes. Each node runs Compute Node Linux (CNL) and has one quad-core AMD Opteron processor and 8 GB of main memory. All nodes are connected in a 3D torus using a HyperTransport link to a Cray Seastar2 engine. The system has a total of 379 TB fiber channel RAID disk space that is managed by a Lustre file system.

6.3 Experiments and Results

6.3.1 Experiments

In this section, the disk I/O times of MADbench2 will be predicted, with a DASH I/O node as the target system and Jade as the reference system. IOR is used to simulate the I/O operations of MADbench2 and the results will be reported

in the next section.

6.3.2 Results

Since wall clock time of an application may differ from run to run, each application was executed five times on the machine and an average run time was used in the above two equations. Thus, the average run times of I/O micro benchmark was used to predict run times of the applications under study. To calculate the prediction error, the predicted time is compared against the average run time of five executions of the applications under study. Accuracy for each target application x is calculated using Equation 6.3.

$$PredictionError_x = 100 * \frac{PredictedTime_x - ActualTime_x}{ActualTime_x} \tag{6.3}$$

In this equation, a negative value indicates that actual I/O time was greater than the predicted time, and a positive value indicates that the actual I/O time was less than the predicted time. For MADbench2 the prediction errors are as follows:

- POSIX API: -20.66%, -8.59%, and -14.79% for reads, writes, and total I/O time respectively
- MPIIO API: -19.92%, -14.76%, and -17.50% for reads, writes, and total I/O time respectively

6.4 Related Work

Related work has pursued I/O modeling and prediction by using script based benchmarks to replay an applications causal I/O behavior [81][82] or using parameterized I/O benchmarks [115] to predict run time on the target system. The modeling approach of this work differs from the related work by using parameterized benchmarks to compute speedup ratios on target systems for each call and use that to predict an applications I/O time.

Pianola [81] is a script based I/O benchmark that captures causal information of I/O calls made by a sequential application. The information is captured by a binary instrumentation tool that, for each call, captures wall clock time of the call, the time spent servicing the call, and arguments passed to the call. Using this information a script is constructed which has sufficient information to replay an application's I/O calls and time between two successive calls. Additionally, the script is also augmented to simulate the memory used by an application between calls. A replay engine can then use this script to replay an applications I/O behavior on any platform.

Like Pianola, TRACE [82] is a script-based I/O benchmark that simulates an I/O behavior of an application using causal information about the I/O calls. Unlike Pianola, TRACE uses interposed I/O calls to capture information regarding I/O calls. TRACE is targeted for parallel applications and hence captures I/O events for each parallel task. In addition to I/O events, for each task, TRACE also includes information related to synchronization delays and computation time. Using this information a replayer simulates the I/O characteristic of each task of the original application.

In [115], IOR was used to simulate the I/O behavior of HPC applications. In this research an application's I/O behavior is first obtained by code and algorithm analysis and this information is then used to prepare inputs for the IOR benchmark. Next, IOR is then run on the target system to predict the actual I/O time of an application.

6.5 Conclusions and Future Work

This chapter presented a methodology to predict disk I/O performance of flash storage systems. The method used a configurable I/O benchmark to measure speedup ratios of each I/O operation of an application and used them to predict an applications total I/O time. The evaluation showed that for large size IO calls reasonable accuracy may be obtained by using this simple model and in the best case the prediction error is only 8.59%.

To further improve the predictions, other parameters such as file caching, contention, and synchronization delays are being investigated. File caching is the ability to cache files in the memory subsystem; file caching can significantly speed up disk I/O. Contention affects the share of I/O resources that each application receives and thus is important to model. Finally, synchronization delays reflect how barrier synchronization and other data dependencies in the application change the rate at which I/O calls are made.

This chapter, in part, is a reprint of the material as it appears in the workshop on Application of Communication Theory to Emerging Memory Technologies (ACTEMT'10) hold with Globecom'10, a joint work with Mitesh Meswani, Pietro Cicotti, and Allan Snavely. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusions

This dissertation presented PMap: a set of methods and tools mapping HPC application performance to a small number of performance primitives. These performance primitives can be extracted manually or automatically using tools provided from real applications. With these small primitives, application performance can be predicted quickly (even on the fly) with reasonable accuracy for aggressive HPC environments, such as dynamic computational grids, complicated hybrid computing platforms, and innovative storage systems.

Performance Characterization and Online Prediction for Computation Grids Performance series were gathered via a deployment of a monitoring and benchmarking infrastructure on two production grid platforms, TeraGrid and Geon. The results showed that these production grids are rather unavailable, with success rates for benchmark and application runs between 55% and 80%. It was found that performance fluctuation was in the 50% range, expectedly mostly due to batch schedulers. This work also investigated whether the execution time of a typical grid application can be predicated based on previous runs of simple benchmarks (performance primitives). Perhaps surprisingly, application execution time can be predicted with a relative error as low as 9%.

Performance Idioms Recognition for Scientific Applications Basic performance primitives called **performance idioms**, such as stream, transpose, reduc-

tion, random access and stencil, are common in scientific numerical applications. It was hypothesized and then proven that a small number of idioms can cover most programming constructs that dominate the execution time of scientific codes and can be used to approximate the application performance. To check these hypotheses, this work proposed an automatic idioms recognition method and implemented the method, based on the open source compiler Open64. With the NAS Parallel Benchmark (NPB) as a case study, the prototype system is about 90% accurate compared with idiom classification by a human expert. The results showed that the above five idioms suffice to cover 100% of the six NPB codes (MG, CG, FT, BT, SP and LU). The performance of the idiom benchmarks with their corresponding instances in the NPB codes were also compared on two different platforms with different methods. The approximation accuracy is up to 96.6%. The contribution is to show that a small set of idioms can cover more complex codes, that idioms can be recognized automatically, and that suitably defined idioms may approximate application performance.

DASH: a Flash-based Data Intensive Supercomputer Data intensive computing can be defined as computation involving large datasets and complicated I/O patterns. Data intensive computing is challenging because there is a five-orders-of-magnitude latency gap between main memory DRAM and spinning hard disks; the result is that an inordinate amount of time in data intensive computing is spent accessing data on disk. To address this problem, a prototype data intensive supercomputer named DASH was designed and built, exploiting flash-based Solid State Drive (SSD) technology and also virtually aggregated DRAM to fill the "latency gap". DASH uses commodity parts including Intel ® X25-E flash drives and distributed shared memory (DSM) software from ScaleMP ®. The system is highly competitive with several commercial offerings by several metrics including achieved IOPS (input output operations per second), IOPS per dollar of system acquisition cost, IOPS per watt during operation, and IOPS per gigabyte (GB) of available storage. This work presented an overview of the design of DASH, an analysis of its cost efficiency, then a detailed recipe for how to design and tune it for high data-performance, lastly showed that running data-intensive scientific

applications from graph theory, biology, and astronomy, as much as two orders-of-magnitude speedup were achieved compared to the same applications run on traditional architectures.

Performance Characterization of Flash Storage System Flash-based SSDs (Solid State Disks) are promising for data intensive HPC applications. However, since all the existing hardware and software were designed without flash in mind, the question is how to integrate the new technology into existing architectures. To explore the potentials and issues of integrating flash into today's HPC systems, a large parameter space was swept by fast and reliable measurements to investigate varying design options. Some lessons learned and also suggestions for future architecture design were provided. The results showed that performance can be improved by as much as $9x$ with appropriate existing technologies and probably further improved by future ones.

Performance Prediction of HPC Applications on Flash Storage System This work extended the PMaC framework to model and predict application performance on flash storage systems. A data-intensive application benchmark called MADbench2 was studied. The results showed that the total I/O time can be predicted with reasonable error of 14.79% for MADbench2.

PMap The end result of this body of work is that the performance of applications on supercomputers can be understood by mapping their performance genetics. It is possible to extract performance primitives reflecting common patterns of computation and data access from real applications manually or automatically. These primitives can then be applied for fast performance modeling and prediction of HPC applications.

Appendix A

TeraGrid Errors

- 0: No valid proxy found
- 1: GRAM Job submission failed because the connection to the server failed (check host and port) (error code 12)
- 2: '[homedir]': Transport endpoint is not connected
- 3: GRAM Job submission failed because the job manager detected an invalid script status (error code 25)
- 4: GRAM Job submission failed because the job manager failed to create the temporary stderr filename (error code 70)
- 5: GRAM Job submission failed because the job manager failed to create the temporary stdout filename (error code 69)
- 6: SoftEnv 1.4.2: updating your software environment, one moment...
- 7: GRAM Job submission failed because data transfer to the server failed (error code 10
- 8: GRAM Job failed because the executable does not exist (error code 5)
- 9: Assertion GLOBUS_FALSE && "listen() failed" failed in file globus_io_tcp.c at line 681 /usr/local/apps/globus-2.4.3-gcc-r5/bin/globus-job-run: line 1: 12738 Aborted

- 10: Network is Unreachable
- 11: No such File or Directory
- 12: gram_init failure: GSS Major Status: General failure GSS Minor Status Error Chain:acquire_cred.c:125: gss_acquire_cred: Error with GSI credential globus_i_gsi_gss_utils
- 13: error: the server sent an error response: 425 425 Can't open data connection.
- 14: GRAM Job failed because the job manager failed to stage the executable (error code 43)
- 15: GRAM Job submission failed because authentication with the remote server failed (error code 7)
- 16: Stale NFS file handle
- 17: GRAM Job submission failed because authentication failed: GSS Major Status: Unexpected Gatekeeper or Service Name GSS Minor Status Error Chain
- 18: GRAM Job submission failed because the cache file could not be opened in order to relocate the user proxy (error code 75)
- 19: an end-of-file was reached
- 20: proxy is not valid long enough
- 21: Error detected on file transfer: error: the server sent an error response: 550 550 /scratch/local/millsc/hpf3.preali.rot.SL: not a plain file.
- XXX: GRAM Job submission failed because authentication failed: GSS Major Status: Unexpected Gatekeeper or Service Name GSS Minor Status Error Chain: init.c:499: globus_gss_assist_

Appendix B

GEON Errors

- Error detected on file transfer: error: globus_xio: System error in connect: Connection timed out globus_xio: A system call failed: Connection timed out
- org.globus.ogsa.impl.base.gram.client.GramJob [setStatusFromServiceData: 1226] ERROR: Error staging RSL element ns1:value [URL] to the GASS
- Invalid ns1:value path
- ERROR utepgeon01.utep.edu: Execution returned non-zero status: 256 100 [09/20/2005 06:18:54:368] org.globus.ogsa.impl.base.gram.client.GramJob [request: 479] ERROR: problem accessing Managed Job service AxisFault faultCode: http://schemas.xmlsoap.org/so at time 2005-09-20 06:18:54

Bibliography

[1] 3Leaf System Inc. `http://www.3leafsystems.com/`.

[2] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. Poems: End-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, 26(11):1027–1048, 2000.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and P. R. Design tradeoffs for ssd performance. pages 57–70. USENIX Annual Technical Conference, 2008.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison Wesley, Pearson Education, Inc., 1986.

[5] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.

[6] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and Z. W. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.

[7] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4, HP Laboratories, Palo Alto, California, July 2001.

[8] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Conference on File and Storage Technology (FAST'02)*, pages 175–188, Jan. 2002.

[9] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression using operand factorization. *Proceedings of the 31th Annual International Symposium on Microarchitecture*, 1998.

[10] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view

from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[11] D. Bader, editor. *Petascale Computing: Algorithms and Applications*. Chapman & Hall/CRC Press.

[12] R. M. Badia, J. Labarta, J. Gimenez, and F. Escale. Dimemas: Predicting mpi applications behavior in grid environments. In *Workshop on Grid Applications and Programming Tools (GGF8)*, 2003.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[14] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transaction of Software Engineering*, 30(5):295–310, MAY 2004.

[15] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[16] External memory bfs code. `http://www.madalgo.au.dk/~ajwani/em_bfs/`.

[17] Biological networks website. `http://biologicalnetworks.net/`.

[18] V. Blanco, J. A. Gonzalez, C. Leon, C. Rodriguez, G. Rodriguez, and M. Printista. Predicting the performance of parallel programs. *Parallel Computing*, 30(3):337–356, 2004.

[19] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. pages 142–153. International symposium on Computer architecture, 1994.

[20] J. Borrill. Madcap-the microwave anisotropy dataset computational analysis package. *Arxiv preprint astro-ph/9911389*, 1999.

[21] J. Borrill, L. Oliker, J. Shalf, H. Shan, and A. Uselton. HPC global file system performance analysis using a scientific-application derived benchmark. *Parallel Computing*, 35(6):358–373, 2009.

[22] T. Brewer. Instruction Set Innovations for Convey's HC-1 Computer. *The 21st Symposium of High Performance Chips (HotChips)*, 2009.

[23] Btrfs project. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[24] RedHat bugzilla: Setting IRQ affinity does not work with MSI devices. https://bugzilla.redhat.com/show_bug.cgi?id=432451.

[25] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report Technical Report 1342, University of Winsconsin-Madison, Computer Science Department, 1997.

[26] L. C. Carrington, A. E. Snavely, M. Laurenzano, R. L. C. Jr., and L. P. Davis. How well can simple metrics represent the performance of hpc applications? In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, NOV 2005.

[27] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[28] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS '09*, pages 217–228, New York, NY, USA, 2009. ACM.

[29] M. Chapman and G. Heiser. vNUMA: A virtual shared-memory multiprocessor. USENIX Annual Technical Conference, 2009.

[30] Chauvenet's criterion page at widipedia. http://en.wikipedia.org/wiki/Chauvenet%27s_criterion.

[31] F. Chen, D. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. pages 181–192. SIGMETRICS/Performance, 2009.

[32] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of workloads used in high performance and technical computing. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 73–82, New York, NY, USA, 2007. ACM.

[33] A. Chien. Does 10x10 replace 90/10, Salishan'10. http://www.lanl.gov/orgs/hpc/salishan/index10.shtml.

[34] G. Chun, H. Dail, H. Casanova, and A. Snavely. Benchmark Probes for Grid Assessment. In *Proceedings of the High-Performance Grid Computing Workshop*, April 2004.

[35] F. Collins, M. Morgan, and A. Patrinos. The human genome project: Lessons from large scale biology. *Science*, 300(5617):286–290, 2003.

[36] W. V. Courtright II, G. A. Gibson, M. Holland, and J. Zelenka. RAIDframe: rapid prototyping for disk arrays. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 268–269, Philadelphia, PA, 1996. ACM Press.

[37] SPEC CPU2006 home page. `http://www.spec.org/cpu2006/`.

[38] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. Eicken. Logp: Towards a realistic model of parallel computation. In *ACM Symp. on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, May 1993.

[39] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[40] Distributed European Infrastructure for Supercomputing Applications. `http://www.deisa.org/`, 2006.

[41] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *The Computer Journal*, 35(2):68–76, Feb. 2002.

[42] EXT4 page at Wikipedia. `http://en.wikipedia.org/wiki/Ext4`.

[43] Sun f5100 technical specification and price information. `http://www.sun.com/storage/disk_systems/sss/f5100/specs.xml`.

[44] Facebook website. `http://www.facebook.com`.

[45] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[46] M. Frumkin. Data flow pattern analysis of scientific applications. In *Workshop on Patterns in High Performance Computing*, May 2005.

[47] M. Frumkin and R. Van der Wijngaart. NAS Grid Benchmarks: A Tool for Grid Space Exploration. *Cluster Computing*, 5(3), 2002.

[48] A. Funk, V. Basili, L. Hochstein, and J. Kepner. Analysis of parallel software development using the relative development time productivity metric. *CT Watch*, 2:4A, 2006.

[49] Fusionio technical specification of 160 gb slc pcie iodrive. http://www.fusionio.com/products/iodrive/?tab=specs.

[50] L. Grupp, A. Caulfield, J. Coburn, E. Yaakobi, S. Swanson, and P. Siegel. Characterizing flash memory: Anomalies, observations, and applications. MICRO, 2009.

[51] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lain. Interprocedural parallelization analysis in SUIF. *ACM Transactions on Programming Languages and Systems*, 27(4):662–731, Jul. 2005.

[52] J. He, J. Bennett, and A. Snavely. DASH-IO: an empirical study of flash-based IO for HPC. In *Proceedings of the 2010 TeraGrid Conference*, TG '10, pages 10:1–10:8, New York, NY, USA, 2010. ACM.

[53] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. DASH: a Recipe for a Flash-based Data Intensive Supercomputer. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[54] J. He, A. E. Snavely, R. F. Van der Wijngaart, and M. A. Frumkin. Code coverage, performance approximation and automatic recognition of idioms in scientific applications. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 223–224, New York, NY, USA, 2008. ACM.

[55] T. Hey and D. Lancaster. The development of Parkbench and performance prediction. *The International Journal of High Performance Computing Applications*, 14(3):205–215, 2000.

[56] HPC Challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[57] The ssd improv: Intel & indilinx get trim, kingston brings intel down to $115. http://www.anandtech.com/storage/showdoc.aspx?i=3667&p=1.

[58] TeraGrid Inca Test Harness and Reporting Framework. http://tech.teragrid.org/inca/.

[59] The Inca Reporter Guide. http://tech.teragrid.org/inca/www/documentation.html.

[60] InfiniBand page at Wikipedia. http://en.wikipedia.org/wiki/InfiniBand.

[61] Intel ® ssd firmware update. http://www.intel.com/go/ssdfirmware.

[62] Intel ® MPI Benchmarks 2.3. http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/219848.htm, 2005.

[63] Ior benchmark. http://sourceforge.net/projects/ior-sio/.

[64] Itanium wikipedia page. http://en.wikipedia.org/wiki/Itanium.

[65] ixbt labs, hdd power consumption and heat dissipation of enterprise hard disk drives. http://ixbtlabs.com/articles2/storage/hddpower-pro.html.

[66] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[67] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. 2001.

[68] O. Khalili, J. He, C. Olschanowsky, A. Snavely, and H. Casanova. Measuring the performance and reliability of production computational grids. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, pages 293–300, Washington, DC, USA, 2006. IEEE Computer Society.

[69] D. Kodeboyina and B. Plale. Experiences with OGSA-DAI: Portlet Access and Benchmark. In *Global Grid Forum Workshop on Designing and Building Grid Services*, September 2003.

[70] P. Kogge, editor. *ExaScale Computing Study: Technology Challenges in Achieving Exascale System*. http://www.sdsc.edu/~allans.

[71] La silla observatory website. http://www.eso.org/sci/facilities/lasilla/.

[72] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for Linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010.

[73] K. Li. Ivy: A shared virtual memory system for parallel computing. pages 94–101. International Conference on Parallel Processing, 1988.

[74] LINPACK Benchmark. http://www.top500.org/lists/linpack.php, 2005.

[75] Large synoptic survey telescope (lsst) website. http://www.lsst.org/lsst/about.

[76] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. *Proceedings of the 12th International Parallel Processing Symposium (IPPS)*, 1998.

[77] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, B. Liu, L. Johnsson, and J. Mellor-Crummey. Scheduling strategies for mapping application workflows onto the grid. In *14th IEEE Symposium on High Performance Distributed Computing (HPDC 2005)*. IEEE Computer Society Press, 2005.

[78] MAPS Benchmark. `http://www.sdsc.edu/PMaC/MAPs`, 2005.

[79] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 2–13. ACM Press, 2004.

[80] M. Mathis, D. Kerbyson, and A. Hoisie. A performance model of nondeterministic particle transport on large-scale systems. In *Proc. Computational Science - ICCS*, 2003.

[81] J. May. Pianola: A script-based I/O benchmark. In *Petascale Data Storage Workshop, 2008. PDSW'08. 3rd*, pages 1–6. IEEE, 2008.

[82] M. Mesnier, M. Wachs, R. Sambasivan, J. Lopez, J. Hendricks, G. Ganger, and D. O'Hallaron. TRACE: Parallel trace replay with approximate causal events. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, page 24. USENIX Association, 2007.

[83] M. Meswani, P. Cicotti, J. He, and A. Snavely. Predicting Disk I/O Time of HPC Applications on Flash Drives. In *Workshop on Application of Communication Theory to Emerging Memory Technologies with Globecom'10*, 2010.

[84] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill. Bit error rate in NAND Flash memories. In *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, pages 9–19. IEEE, 2008.

[85] The Message Passing Interface (MPI) standard. `http://www-unix.mcs.anl.gov/mpi/`.

[86] Message Signaled Interrupts page at Wikipedia. `http://en.wikipedia.org/wiki/Message_Signaled_Interrupts`.

[87] S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[88] National Center for Microscopy and Imaging Research. http://ncmir.ucsd.edu/, 2005.

[89] Ncq page at wikipedia. http://en.wikipedia.org/wiki/Native_Command_Queuing.

[90] Neuroscience information framework (nif). http://nif.nih.gov/.

[91] NSF Middleware Initiative Release 7. http://www.nsf-middleware.org/Lists/NMIR7/AllItems.aspx.

[92] M. Norman and A. Snavely. Accelerating data-intensive science with Gordon and Dash. In *Proceedings of the 2010 TeraGrid Conference*, pages 1–7. ACM, 2010.

[93] J. Odom, J. K. Hollingsworth, L. DeRose, K. Ekanadham, and S. Sbaraglia. Using dynamic tracing sampling to measure long running programs. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 59, Washington, DC, USA, 2005. IEEE Computer Society.

[94] D. Ofelt and J. L. Hennessy. Efficient Performance Prediction for Modern Microprocessors. In *Proceedings of ACM SIGMETRICS 2000*, pages 229–239, June 2000.

[95] R. Oldfield, S. Arunagiri, P. Teller, S. Seelam, M. Varela, R. Riesen, and P. Roth. Modeling the impact of checkpoints on next-generation systems. In *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, pages 30–46. IEEE, 2007.

[96] Open64 compiler. http://www.open64.net/.

[97] Open Research Compiler. http://ipf-orc.sourceforge.net/.

[98] S. Park and K. Shen. A performance evaluation of scientific i/o workloads on flash-based ssds. Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS'09), 2009.

[99] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *SC—03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Nov. 2003.

[100] B. Plale, C. Jacobs, Y. Liu, C. Moad, R. Parab, and P. Vaidya. Benchmark Details of Synthetic Database Benchmark/Workload for Grid Resource Information. Technical Report TR-583, Dept. of Computer Science, University of Indiana, August 2003.

[101] PMaC Convolver. http://www.sdsc.edu/PMaC/projects/convolver.html.

[102] PMaC Prediction Framework. http://www.sdsc.edu/PMaC/projects/index.html.

[103] PMaC Lab. http://www.sdsc.edu/PMaC/.

[104] MultiMAPS Benchmark. http://www.sdsc.edu/PMaC/projects/mmaps.html.

[105] PEBIL. http://www.sdsc.edu/PMaC/projects/pebil.html.

[106] PSiNS. http://www.sdsc.edu/PMaC/projects/psins.html.

[107] PMaCinst. http://www.sdsc.edu/PMaC/projects/pmacinst.html.

[108] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *International Conference on Supercomputing*, 1995.

[109] Power4 wikipedia page. http://en.wikipedia.org/wiki/POWER4.

[110] S. Prakash and R. Bagrodia. MPI-SIM: Using parallel simulation to evaluate MPI programs. In *Winter Simulation Conference*, pages 467–474, 1998.

[111] Palomar transient factory (ptf) website. http://www.astro.caltech.edu/ptf/.

[112] A. Rifkin and B. L. Massingill. Performance analysis for archetypes. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, 1998.

[113] P. Sassone and D. Wills. On the extraction and analysis of prevalent dataflow patterns. *Workload Characterization, 2004. WWC-7. 2004 IEEE International Workshop on*, pages 11–18, 2004.

[114] ScaleMP Inc. http://www.scalemp.com/.

[115] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[116] A. K. Sinha, B. Ludaescher, B. Brodaric, C. Baru, D. Seber, A. Snoke, and C. Barnes. GEON: Developing the Cyberinfrastructure for the Earth Sciences - A Workshop Report on Intrusive Igneous Rocks, Wilson Cycle and Concept Spaces. http://www.geongrid.org/workshops/conceptspace/igneous_rocks/workshop_report_intrusive_igneous_rocks.pdf, 2004.

[117] SKaMPI-Benchmark. http://liinwww.ira.uka.de/~skampi/, 2005.

[118] W. Smith, V. Taylor, and F. I. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance . In *Source Lecture Notes In Computer Science; Proceedings of the Job Scheduling Strategies for Parallel Processing table of contents*, volume 1659, pages 202–219, 1999.

[119] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In *Proceedings of Supercomputing*, November 2002.

[120] A. Snavely, X. Gao, C. Lee, L. Carrington, N. Wolter, J. Labarta, J. Gimenez, and P. Jones. Performance Modeling of HPC Applications. In *PARCO*, pages 777–784, 2003.

[121] A. Snavely, X. Gao, C. Lee, N. Wolter, J. Labarta, J. Gimenez, and J. P. Performance modeling of hpc applications. ParCo, 2003.

[122] A. Snavely, G. Jacobs, and D. A. Bader, editors. *Workshop Report: Petascale Computing in the Biological Sciences*. http://www.sdsc.edu/~allans.

[123] A. Snavely, R. Pennington, and R. Loft, editors. *Workshop Report: Petascale Computing in the Geosciences*. http://www.sdsc.edu/~allans.

[124] The SPEC Benchmarks. http://www.specbench.org.

[125] D. P. Spooner, J. Cao, S. A. Jarvis, L. He, and G. R. Nudd. Performance-aware workflow management for grid computing. *The Computer Journal*, 48(3):347–357, May 2005.

[126] STREAM: Measuring Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/, 1995.

[127] Stxxl project. http://stxxl.sourceforge.net/.

[128] Overview of recent supercomputers 2009. http://www.nwo.nl/nwohome.nsf/pages/NWOA_F7X8HXB_FEng.

[129] A. Szalay and J. Gray. Science in an exponential world. *Nature*, 440:413–414, 2006.

[130] V. Taylor, X. Wu, J. Geisler, and R. Stevens. Using kernel couplings to predict parallel application performance. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 125, Washington, DC, USA, 2002. IEEE Computer Society.

[131] V. Taylor, X. Wu, and R. Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, 2003.

[132] The TeraGrid Project. `http://www.teragrid.org`.

[133] Top500 (Nov. 2010). `http://www.top500.org/lists/2010/11`.

[134] TRIM page at Wikepedia. `http://en.wikipedia.org/wiki/TRIM`.

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

[136] K.-Y. Wang. Precise compile-time performance prediction for superscalar-based computers. In *PLDI*, pages 73–84, 1994.

[137] J. Weinberg and A. Snavely. User-guided symbiotic space-sharing of real workloads. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 345–352, New York, NY, USA, 2006. ACM.

[138] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[139] Intel ® x25-e datasheet and technical documents. `http://download.intel.com/design/flash/nand/extreme/extreme-sata-ssd-datasheet.pdf` and `http://www.intel.com/design/flash/nand/extreme/technicaldocuments.htm`.

[140] Tom's hardware, intel ®'s x25-m solid state drive reviewed. `http://www.tomshardware.com/reviews/Intel-x25-m-SSD,2012-13.html`.

[141] XDD benchmark, version 6.5. `http://www.ioperformance.com/`, retrieved in September 2009.

[142] XFS project. `http://oss.sgi.com/projects/xfs/`.

[143] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *11th International Symposium on High Performance Computer Architecture (HPDC-11)*, 2005.

[144] D. York. The sloan digital sky survey. *Astronomical Journal*, 2000.