Stony Brook University

Harnessing Multicore Parallelism for High Performance Data Replication

A Dissertation presented

by

Tan Li

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

December 2015

Stony Brook University

The Graduate School

Tan Li

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dantong Yu - Dissertation Advisor
Adjunct Professor, Department of Electrical and Computer Engineering

Wendy Tang - Chairperson of Defense
Associate Professor, Department of Electrical and Computer Engineering

Shudong Jin
Adjunct Professor, Department of Electrical and Computer Engineering

Mike Ferdman
Assistant Professor, Department of Computer Science

Robert J. Harrison
Professor and Endowed Director of Institute for Advanced Computational Science

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Harnessing Multicore Parallelism for High Performance Data Transfer

by

Tan Li

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

2015

High speed data replication is vital to the data intensive scientific computing that often requires transferring large volumes of observation and simulation data efficiently among geographically dispersed facilities. Enterprise cloud services also depend on effective data replication to scale out of their own data centers, and to optimize work completed per dollar invested. In addition, end consumers are also extremely sensitive to the latency and responsiveness to share data within the network of friends, web-based services and their personal mobile devices. Recent hardware advances offer the opportunity to enable ultra high-speed data replication by aggressively adding more CPU cores, higher network bandwidth and faster

storage into a single commodity server. However, existing monolithic software is designed for the architecture in the past several decades, and not capable of mitigating the system I/O bottlenecks. New effective resource scheduling algorithms and software designs are indispensable to match the I/O bare-metal capability with the actual application's performance.

Designing an end-to-end efficient solution for high speed data replication is non-trivial because of a variety of interconnected factors: 1) the optimal resource scheduling on multicore system is proven to be NP-hard, and even the heuristic algorithm incurs prohibitive computation cost; 2) data replication involves many components along the end-to-end I/O paths, including PCI buses, CPU interconnect links, various I/O controllers and chipsets, each of which can potentially become a performance bottleneck; 3) heterogeneous I/O devices might demonstrate different system performance under various workloads and access patterns. It is up to data replication applications to choose and apply appropriate optimization techniques and to adjust their access patterns to achieve maximum system performance; and 4) the requirement of scaling up to many requests and users necessitates the maximal exploitation of the system parallelism and concurrency that are available in state-of-the-art computer architecture. In conclusion, these challenges give rise to the significant effort to rethink, redesign and re-implement the entire software suite.

We first analyze the state-of-the-art I/O devices and multi-core systems by using benchmark tools and monitoring various performance event counters. In addition, we propose a new metric (i.e., the NUMA scheduling factor) and a performance modeling method to describe the accurate I/O performance patterns and to guide the downstream resource scheduling and mapping. Based on our findings, we propose a variety of mathematical and empirical resource scheduling methods to improve the overall system performance. We model the resource mapping for end-to-end data replication

as a min-sum-max resource allocation problem (MSMRAP), prove its prohibitive computation complexity, and give possible solutions. Finally, we integrate the proposed optimization strategies into a complete data replication solution that employs the asynchronous programming paradigm and supports the resources-aware task scheduling and data preprocessing to maximize the capacity of state-of-the-art hardware systems. The evaluation results obtained from a fully-featured WAN network testbed confirm the effectiveness and remarkable performance advantages of our proposed software system for a comprehensive set of workloads, i.e., 28%- 160% higher bandwidth for transferring large files, a factor of 1.7x-66x speed-up for small files, and up to 108% more throughput for mixed workloads, compared to the widely adopted tools, *GridFTP*, *BBCP* and *Aspera*.

This dissertation leverages the large body of multicore research already accomplished within the HPC community and implement multicore-aware schedulers to improve processor, memory, and I/O affinities for individual tasks (file caching, compression, encryption, and network transport) involved in the end-to-end data replication. Traditional synchronous processing, storage I/O, and network send/receive, even easy to implement, become bottlenecks in harnessing multi-/many-core architectures. Asynchronous operations, commonly found in RDMA, advanced storage I/O, and exascale computing, demonstrate their superior performance and great flexibility over their synchronous counterparts. We designed an asynchronous high throughput data replication system for multicore/many-core computer platforms to allow users to plug in comprehensive libraries for data compression, encryption, transformation, and checksum for different processing environments. This dissertation paves a way for advancing large scale data transfers in excess of 100 Gbps, and bridging the gap between the bare metal network performance and effective end-to-end data transfer capability. The expected research outcomes will have preeminent visibility

in high-speed networks, data management middleware, cloud computing, and exascale supercomputing.

- To my beloved wife, Wenjing Zhang, and my parents

Contents

List of Figures

List of Tables

Acknowledgements

I am deeply appreciative of the guidance of my committee members, the help from my follow students, and the support from my wife and parents.

Above all, I would like to express my deepest gratitude to my adviser, Dantong Yu, for inspirational and timely advice and constant encouragement throughout my PhD study. I have learned a great deal from his unique perspective on research and his personal integrity and expectations of excellence. I would also like to thank my co-adviser, Shudong Jin for for guiding my research, patiently correcting my paper writing, and helping me to develop my academic background. I really appreciate the vital supports from both of my advisers. I am very fortunate to have had Wendy Tang, Mike Ferdman and Robert Harrison on my committee. Special thanks to them for sparing their precious time to help me with the dissertation and defense.

I am indebted to other faculty and staff members in the Department of Electrical and Computer Engineering. I want to thank the Graduate Director, Yuanyuan Yang, for introducing me into the department. Thank Rachel Ingrassia and Susan Hayden for navigating the steps to TA/RA paperwork, applying for jobs, completing the dissertation, and graduating.

My fellow students' support has been continuous fuel during my long journey in finishing this doctoral program. I would like to thank the research group members, Yufei Ren, Zhenzhou Peng and Shun Yao. Special thank also goes to the ESNET supporting team, Mellanox Technologies, LSI Corporation and Fusion-IO Inc for their equipment donations. Thank you to everyone who has made this dissertation possible.

Publications

Journal Publications

- Tan Li, Yufei Ren, Dantong Yu, Shudong Jin, "RAMSYS: Resource-Aware Asynchronous Data Replication with Multicore SYStems", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, *under review*.

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Design, Implementation, and Evaluation of a NUMA-Aware Cache for iSCSI Storage Servers", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol.26, no. 2, pp. 413-422, Feb. 2015, doi:10.1109/TPDS.2014.2311817.

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Design and Testbed Evaluation of RDMA-Based Middleware for High-Performance Data Transfer Applications", *Journal of Systems and Software*, Volume 86, Issue 7, July 2013, Pages 1850-1863, ISSN 0164-1212, 10.1016/j.jss.2013.01.070.

Conference Publications

- Tan Li, Yufei Ren, Dantong Yu, Shudong Jin, "Resources-conscious Asynchronous High-speed Data Transfer in Multicore Systems: Design, Optimizations, and Evaluation", In *Proceedings of Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*

, vol., no., pp.1097,1106, 25-29 May 2015 28th International, (IPDPS '15), May, 2015.

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Design and Performance Evaluation of NUMA-Aware RDMA-Based End-to-End Data Transfer Systems", In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, (SC '13)*, Denver, Colorado, November 2013.

- Tan Li, Yufei Ren, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Characterization of Input/Output Bandwidth Performance Models in NUMA Architecture for Data Intensive Applications", In *Proceedings of the International Conference on Parallel Processing, (ICPP '13)*, Lyon, France, October 2013.

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, Brian L. Tierney, Eric Pouyoul, "Protocols for Wide-Area Data-intensive Applications: Design and Performance Issues", In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, (SC '12)* , Salt Lake City, Utah, November 2012.

Workshop Publications

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Middleware Support for RDMA-based Data Transfer in Cloud Computing", In *Proceedings of the High-Performance Grid and Cloud Computing Workshop*, Shanghai, China, May 2012.

Chapter 1

Introduction

In this chapter, we present the motivations of the research in the context of the ever-growing hardware capacity and identify the multiple challenges in multicore-aware high-speed data replication. Subsequently, we offer the main contributions of the dissertation to resolve these challenges.

1.1 Motivation

Data intensive applications often adopt high performance replication tools to reduce the data distribution time and the overall computation cost: 1) Scientific applications, which mainly aim to answer the most fundamental questions facing human society, are generating increasingly large amount of data that must be distributed, accessed, and analyzed by scientists collaborations worldwide. Furthermore, these collaborations often coordinate hundreds of geographically distributed computation, storage, and networking resources to share, manage, and process big data for accelerated science discovery. For example, the Brookhaven National Lab (BNL) participates into the ATLAS experiment at the European Large Hadron Collider (LHC) near Geneva, in Switzerland that produces petabytes of raw and processed data every year and involves $3,000$ particle physicists around the world [1].

Effective data replication mechanism is indispensable to support the collaboration and data processing of such a scale. 2) Companies, such as Amazon, Facebook, Google and Microsoft, are making significant investments in extremely large scale data centers and providing cloud services. Valuable enterprise data in those clouds have a number of formats and sizes: for example, relational databases, files, web pages, and packaged applications. Fast and reliable moving these data regardless their size and formats among different data centers becomes vital to their business success for virtually all industries. It is economies-of-scale to amortize hardware expense by accommodating as many user requests as possible and maximizing the aggregate input/output (I/O) performance from dedicated storage and network resources. 3) An increasing number of individual consumers utilize public cloud service to store, manage and share their personal data, including documents, images, photos and videos. The latency and responsiveness of uploading, downloading and syncing their data largely affect user experience and satisfaction [2]. Highly efficient data replication is a widely-adopted strategy to ensure the high Quality of Service (QoS) and to increase the concurrency of cloud services.

On the other hand, the recent advancements in computer hardware equip server systems with a large number of cores, a deep memory hierarchy, and ultra high-speed input/output (I/O). It essentially removes many hardware capability constraints and provides an opportunity for next generation data replication services. Among these advance computer architectures, multi-socket systems with Non-Uniform Memory Access (NUMA) became prevalent in a wide spectrum of computing platforms, from desktop computers, high-end servers to the computing nodes in data center clusters and supercomputers. Each socket has dedicated on-chip modules that connect with various hardware components, i.e. memory controllers, a portion of system cache, interrupt controllers, and other peripheral devices.

Main memory and peripheral interfaces are distributed among processor nodes. Scalable interconnect technologies, such as Intel's QuickPath Interconnect (QPI) and AMD's HyperTransport (HT), link all nodes with high-speed communication channels. These architectures, compared to their homogeneous peers, provide higher performance at a much lower cost in terms of die area and power consumption [3, 4]. Therefore, even though individual CPU chips/cores have moderate performance improvement over their prior generations, the introduction of the NUMA architecture still renders a Moore's Law growth in the aggregate capacity of the new processors. Furthermore, memory subsystem and inter-node bus has been improving rapidly in the last decade. For example, the Intel Xeon Sandy Bridge aggregates four DDR3 memory channels and two QPI interfaces and delivers 51.2 Gbyte/s maximum memory bandwidth and 128 Gbyte/s QPI bandwidth. Lastly, the capability of network and storage I/O devices are also growing dramatically. 40 Gbps Ethernet adapters has already been available to commodity servers for the past five years. The line rate of new Ethernet reaches 100 Gbps in the market of high-end server adapters, switches and routers [5]. In addition, solid-state drive array delivers more than 80 Gbps I/O bandwidth to a single host [6]. Currently high-end servers and gateway nodes are often equipped with multiple network adapters and large disk arrays to provide abundant aggregate bandwidth to parallel data transfers.

However, existing data replication methodology and software fail to utilize the hardware capabilities effectively to improve the application-perceived or user-perceived performance. The biggest cloud service vendor, Amazon, employs a compromised solution to copy user data in a storage drive, and sends it via UPS [7]. IBM acquired a successful commercial solution, Aspera, that only scales to 1/10 Gbps networks [8]. We summarize the reasons for the suboptimal performance as follows:

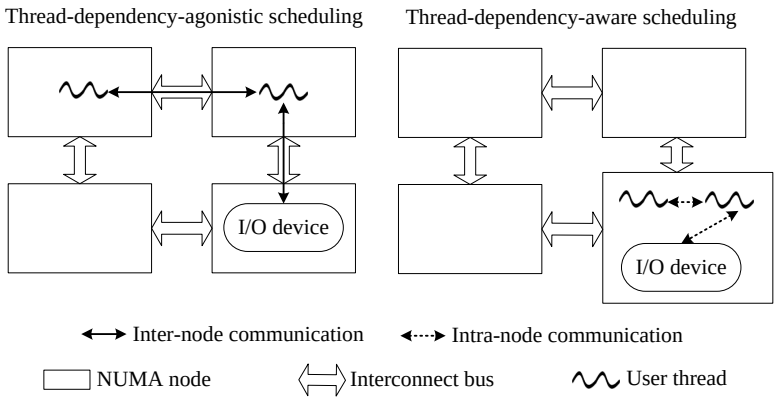


Figure 1-1: Thread-dependency-agnostic versus thread-dependency-aware scheduling in a four-node NUMA system

1.1.1 Poor NUMA-aware support

Existing data replication applications rely on the default OS behavior which often adopts a thread-dependency-agnostic scheduling, i.e., it dynamically chooses a CPU to run a user thread, and potentially migrates it later to another NUMA node to balance the overall system load. However, thread scheduling and migration that ignore the characteristics of NUMA architecture undermine the performance of I/O-intensive applications. Hence, instead of using default OS scheduling, high-speed data replication software must resort to implementing its own thread-dependency-aware scheduler. As shown in Figure 1-1, an applications own scheduler has more knowledge about its data access patterns and I/O devices in use, and thereby can schedule all its threads and memory to the NUMA node that is closely aligned with its target devices. Consequently, the scheduler reduces the overhead of inter-node communication, and improves both the applications performance and the systems overall efficiency.

A plethora of research efforts has been trying to enable NUMA-awareness by benchmarking and modeling performance. Most of the current performance models and resource assignment algorithms for NUMA architecture [9–11] are based on hop-distance, directly or indirectly. It is one of the

most popular NUMA metrics, and represents the number of physical links along the data access path between two devices. More hops on this path usually imply a higher accessing cost. However, hop-distance is not a good indicator to the NUMA penalty, especially for I/O performance, because of the following reasons:

- The architectural details of many modern NUMA systems often are not intuitive to users. In [12], the AMD architecture designers illustrated three possible topologies of the 4P AMD Opteron Magny Cours platform. These topologies are shown in Figure 1-2(a,b,c). For the same type of 4P processors, the authors of [13] described another topology variant as Figure 1-2(d). The exact design of a NUMA system depends on the choices that are made by system architects and is therefore implementation-specific, even for the same technology specification. The different link width between the nodes with the same number of hops also significantly impact the NUMA penalty, as shown in research [14].

- Even if the full topological details are known in advance, hop-distance is still not an accurate metric for actual data transfers among CPU, memory, and I/O modules. Maximizing data locality does not always minimize the execution time of data intensive applications, especially in a multi-user/multi-task cluster environment. The authors of [15] demonstrated, for their eight-node AMD host, an overhead because of the cache coherency traffic led to a higher penalty to cores on the edges compared to the middle ones in the host topology. Even local memory access has a significant performance difference over CPU nodes within the same CPU socket.

- The work in [16] indicated that the Intel Nehalem and Westmere platforms take fairness into consideration and reserve a large fraction

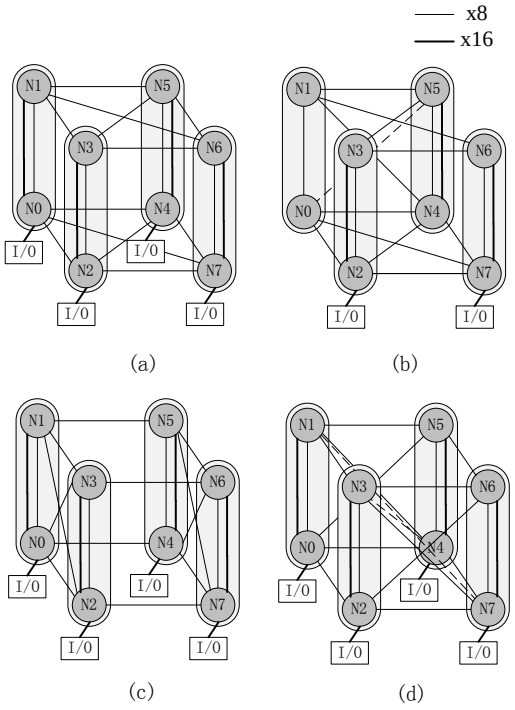


Figure 1-2: Possible topologies of 4P AMD Opteron Magny Cours Processors. The oval circles in the plot represent the NUMA nodes, including both CPU and memory. The lines between them denote high-speed HyperTransport (HT) interconnections. The width of a link can be 8 or 16 bits.

of the node resources specially for remote memory accesses. Hence, allocating all tasks locally cannot necessarily take the full advantage of the aggregate system memory bandwidth. Moreover, excessive use of local resources will introduce contention and congestion among concurrent tasks on shared queues and buses, and thus degrade the overall system performance [17].

Compared to hop-distance, an accurate NUMA cost model is more suitable to represent every detailed aspect of the system performance features. STREAM benchmark [18] is the most widely adopted tool to understand and model the NUMA penalty to memory accesses. However, as we will present later in Section 4.2, the STREAM benchmark can neither reveal the precise attributes of the I/O bandwidth performance pattern on a given

architecture, nor can it explain the enormous NUMA penalty to the remote I/O adapter accesses. Therefore, new methodologies are indispensable to understand the underlying reasons of the NUMA penalty and accurately model the I/O access cost on a multicore system.

1.1.2 Outdated design of existing data replication software

Although there have been numerous attempts to take advantage of multi-threading and multi-streaming techniques, the existing software designs still lack efficiency. 1) The Linux *scp* and *FTP* are single-threaded/single-stream tools that run only on a single CPU core, and a single instance of these tools does not scale to multiple cores. 2) *GridFTP* [19], a popular data replication tool in scientific computing, uses a single thread pool to handle all network and storage I/Os and to control events. This arrangement leads to frequent context switching and cache re-warming, and is also hard to scale to multicore systems. 3) *BBCP* software [20], another widely adopted tool in the Department of Energy, employs a single thread for storage I/O, regardless of the characteristics of its involved storage device. 4) *UDT* tool [21] builds on the UDP protocol, and suffers from excessive user-level control overheads. It also uses a single thread for all storage I/Os. 5) Aspera [8] is a widely-used UDP-based commercial tool, targets gigabit performance, and does not scale well to the bulk data transfer in the 40/100 Gbps networks. 6) Our previous work on the RFTP tool [22] depends on specific RDMA hardware to offload data transfer tasks to RDMA I/O devices. More importantly, these tools do not incorporate the asymmetrical NUMA architecture, and cannot attain optimal performance.

Another ineffectiveness of existing data replication protocols is that they often treat a single file as an individual transaction that spends sev-

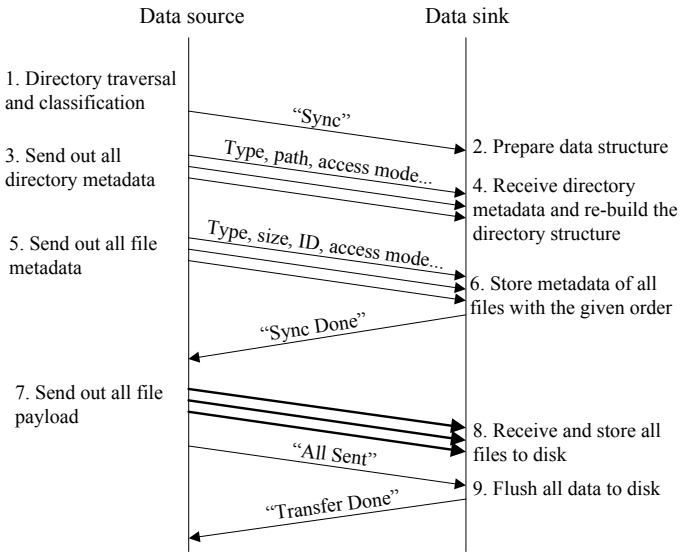


Figure 1-3: Multi-file transfer protocol in FTP and BBCP tool

eral round trips to synchronize each file metadata with the remote peer. Figure 1-3 exemplifies this process with the protocol of transferring multiple files in FTP and BBCP software [20]. For transferring large files, the latency and overheads of processing meta data are relatively small comparing to those for transferring the actual data payloads. However, this design leads to excessive overheads in the case of moving a massive number of small files, especially across high-latency wide-area networks. The scenarios become even more challenging for real workloads that are mixed with large and small files.

1.2 Challenges

While multicore processors and I/O subsystems provide the necessary physical performance, designing and implementing efficient data replication is quite challenging and often complicated by the heterogeneity with the hardware and software components involved along the end-to-end data replication path. Figure 1-4 provides a comprehensive view of relevant research

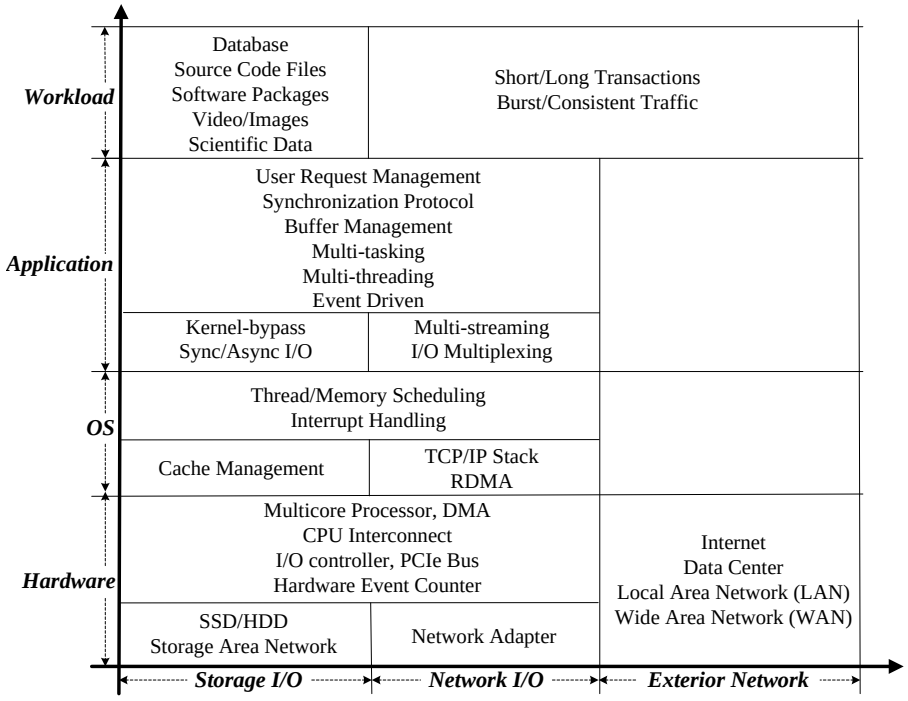


Figure 1-4: Research fields at various system levels along end-to-end data replication path

areas that reside in multiple system levels. A highly efficient data replication pipeline requires smoothly interaction among those modules, and any defect can potentially cause considerable deterioration in performance. This section details the challenges of our research bottom-up as follows:

1) **In-depth understanding of NUMA effects and I/O access patterns.** An in-depth understanding of NUMA effects and I/O access patterns requires comprehensively and thoroughly characterizing every aspect of multicore system. The most popular approaches include hardware event monitoring and I/O benchmarking. Various hardware counters are used to keep track of the events (for example, cache misses, cache stall cycles, floating-point operations, branch mispredictions, CPU cycles, and instructions executed) and are commonly incorporated into the task-concerning software systems and applications, including adaptive CPU scheduling, performance monitoring/debugging, workload pattern identi-

fication, and adaptive applications that manage and schedule the allocated resources by themselves [23]. However, the number of these hardware events is excessive, and their naming, implication and availability vary wildly from one specification and OS to others. An appropriate screening and utilization of these hardware counters demands a good understanding to the system and tedious characterization efforts. On the other hand, many research efforts resort to benchmarking every case of I/O access and consequently require accessing expensive high-speed I/O devices, tuning I/O configurations, and attaining extra high-end server for network tests. The approach often incurs prohibitive cost as well as onerous workloads, especially for the high-node-count (HNC) systems with many I/O hardware components. On the other hand, it is more complicated to measure I/O bandwidth than to memory bandwidth because the actual usable bandwidth of peripherals depends on all resources along the data path between the I/O device and its requesting processor. These resources include PCIe buses, CPU interconnect links, various I/O controllers and chipsets, and the specific PCIe devices (SSD, HDD, Storage Area Network, Network Adapter, and etc.) that contain the requested data as shown in the hardware level of Figure 1-4. The performance bottleneck can potentially reside in any of them.

2)**Inefficient general-purpose OS scheduler fail to recognize the inter-dependency among threads.** The general-purpose OS scheduler often ignores NUMA alignment, and performs application-agnostic thread binding. This strategy leads to excessive inefficient I/O and memory accesses and thereby suboptimal performance in high-throughput computing. Instead of relying on default OS behavior, high performance data replication applications must implement their own thread-dependency-aware resource scheduler. However, thread/memory scheduling on multicore system is already known to be a challenging problem. The NUMA effect and

the requirement to mitigate it will incur further complexity to the scheduling algorithm. Any approach must entail two aspects: 1) *Online dynamic scheduling.* This method requires complicated logic and frequent system profiling and computation to determine the optimal resource allocation and assignment. The overhead is often non-negligible for a large number of small I/O transactions, each of which has a small duration. 2) *Offline performance modeling.* This aspect usually uses benchmark tools to capture the characteristics of system performance, and then utilizes the generated model to guide the future resource scheduling. The intuitive approach for data replication applications is to individually benchmark every subsystem involved in a memory or PCIe access with all possible NUMA configurations. This leads to the similar aforementioned problem in the Section 1.2. Both aspects will be covered in this dissertation. Besides, a highly efficient resource scheduling also needs to consider I/O interrupt binding and cache management.

3)**Scalable application design to maximize system parallelism.** Modern high-end systems often have multiple host adapters for both network communication and external storages. Each type of I/O operations and its associated adapters may exhibit unique behavior under various access patterns, e.g., kernel-bypass, sync/async storage I/O, network multi-streaming and I/O multiplexing. To obtain the maximum aggregate performance, data replication software needs efficient task/buffer management and parallel access to those resources. For this purpose, the software must incorporate new paradigm and design framework, e.g., multi-threading, multi-tasking, event driven, and buffer reuse, that are more difficult and error-prone than the traditional ones, to assure a high-speed end-to-end replication pipeline. Meanwhile, a light-weight protocol is also required to negotiate resource assignment and data replication parameters to support multiple concurrent requests while ensure high performance for each

individual task. At last, the data replication applications need to supply various modules, such as progress reporting and error handling, to interact with users and enhance user experience.

4)**Consistent high performance across various workloads.** Delivering a large volume of content, e.g., scientific data and high-resolution images and videos, consumes a large fraction of bandwidth among all applications in the Internet today. Most studies in high throughput computing community have focused on efficient storage, transfer, and management of large files. However, many production workloads and the majority of consumers primarily manipulates a large number of small files that range from several hundreds of bytes to megabyte. For example, a standard cloud application may need to install complex software packages, OS kernels and databases that consist of thousands of executables, dynamic libraries, and configuration files. The package and database must either be accessed at run-time over networks, which results in many small network transaction, or be installed on a worker node, resulting in a large number of small disk I/Os. Unfortunately, the data throughput of small file transfers is often several orders of magnitude worse than that of bulk transfers of large files [24]. Achieving high performance for transferring large numbers of small files necessitates two fundamental changes to todays solutions, 1) an effective protocol to process and synchronize file metadata between data source and sink using as few message exchanges as possible, and 2) an efficient pipeline to transfer actual payloads. The capability to handle burst I/O traffic and different network latencies are other challenging issues that need to be addressed in data replication design.

1.3 Research Contribution

This dissertation proposes a new system for high performance data replication. We follow a streamlined approach to design and implement a comprehensive system, i.e. analyzing system components' performance, identifying the areas of improvement, using the quantitative evaluation results to guide the system integration, and make the contributions as follows:

1)**In-depth analysis of the NUMA effects.** Chapter 3 quantifies the NUMA penalty and provides a first-order analysis on the NUMA effects of modern high-performance systems. We select and monitor various relevant hardware counters while utilizing intensive memory access and I/O tasks to mimic real world data replications. To the best of our knowledge, no existing study attempts to undertake such a a quantitative evaluation effort. The test results expose the significant impacts of the hardware prefetching contention and the cache coherence traffic inside NUMA systems. These factors lead to a performance degradation of 53% to 85% in memory access and 26% to 60% in network communication. Therefore, we conclude that the bare-metal hardware enhancement cannot mitigate the NUMA penalty and guarantee a better performance for the current data replication applications, and confirm that the NUMA-awareness is vital to achieve superior I/O performance. Meanwhile, we propose a new NUMA metric, i.e., NUMA scheduling factor, to quantify the NUMA penalty in a modern multi-core system.

2)**New methodology and tool for NUMA-aware resource scheduling.** Chapter 4 shows the limitations of existing characterization methods, including the hop-distance to measure the NUMA penalty and the well-known STREAM benchmark [18] to analyze I/O performance. We then design and develop, to the best of our knowledge, the first NUMA characterization software for bulk data I/O tasks. It is capable of characterizing

and predicting the relative bandwidth performance levels among various NUMA configurations without even involving the actual I/O devices and daunting benchmarking. The method is validated by comparing the generated model with the real performance result of various I/O operations, including TCP, Remote Direct Memory Access (RDMA) and disk read/write, in node-level. We conclude that our NUMA characterization method and tool can attain accurate offline I/O bandwidth performance models and help to improve applications' I/O behavior and to assist runtime schedulers on all NUMA platforms.

3)**Mathematical description of multicore thread scheduling problem and preliminary implementation.** We model the multicore thread scheduling for the end-to-end data replication as a staged bipartite graph and explore its potential mathematical solutions in Chapter 5. The computation complexity of this graph model is proven to be NP-hard, even for the relaxed form of the model. Thereafter, we resort to solve the problem empirically. We develop and implement a thread scheduling module for a bulk data transfer software, BBCP. With this module extension, BBCP can specify its affinity preference of thread-to-CPU v.s. thread-to-memory along the entire data path for both client and server processes at runtime. We then thoroughly evaluate the NUMA-aware BBCP over a high-speed network testbed. The experimental results show that the NUMA-aware BBCP consistently runs at the wire speed in the selected testbed system and obtains 10% to 220% bandwidth improvement over the standard BBCP in memory-based tests. It also achieves remarkable performance benefits in a storage-area network testbed.

4)**A complete new design and implementation for the next generation high-speed data replication.** We integrate the aforementioned new findings into a novel design of high-speed data replication software in Chapter 6. Our new design contains a variety of high performance features,

in addition to NUMA-awareness, to ensures a superior performance for different types of workload and multicore systems. Our proposed framework adopts a resource-aware approach to pre-allocate I/O threads, and ensures that resources are reusable and optimally allocated among multiple users and data transfer requests. It also provides storage-centric task mapping and NUMA-aware thread scheduling to ensure the affinitive data movement and communications in multicore systems. The framework design also seamlessly integrates multiple strategies of optimizations, such as file-level sorting, block-level asynchronism, and thread-level pipelining. Finally we provide a reference implementation, termed RAMSYS–Resource-Aware Asynchronous Data Replication with Multicore SYStem. We compare comprehensively our system with the state-of-the-art *GridFTP*, *BBCP* and *Aspera* software, using full-scale high bandwidth network testbeds in both local data center and nationwide long-haul networks. The evaluation over various realistic workloads confirms that our software achieves 1.13x to 1236x speed-up over the other software tools for these workloads.

1.4 Significance and Broader Impacts

Data replication is a widely used technology to copy data over a computer network to one or more remote locations. This research will greatly improve data replication performance, and thus yield substantial benefits to other areas in high performance computing and even to the entire society. (1) The integrated data replication system can be potentially incorporated into several scientific applications of national priorities. For example emergency response and preparedness demand expedited data transfer to the emergency center. Clean energy, bioinformatics and photon science replicate data from experimental facilities to the data processing centers. Exascale simulations constantly analyze petabytes of their simulation results, and

need to move data off-supercomputers that are not necessarily optimized for data-intensive high throughput computing. 2) In the business world, ensuring data security and keeping copies of the data at physically separate locations is a common practice to safeguarding data during a natural disaster or extreme event. Furthermore, improving data replication efficiency can largely save the time and increase the ROI (return of investment) of the valuable data and their supporting infrastructure, such as enterprise data centers. 3) We expect our research prototype eventually utilizes the web service interface so that end consumers can control and move data with their mobile devices. This will help individual consumers to utilize reliable and cost-effective cloud storages and data sharing services for their even-growing personal data.

1.5 Dissertation Overview

The remainder of this paper is organized as follows. Chapter 2 contains the background and design consideratons that are relevant to this work. We conduct extensive experiments to quantify the NUMA remote access cost using iperf network benchmark [25], STREAM memory benchmark and hardware event counters in Chapter 3. Chapter 4 focuses on demonstrating the ineffectiveness of the existing NUMA benchmark methods, describing the design of our empirical methodology, and offering the details of how to apply it to real applications. Chapter 5 provides the mathematical formulation of resource scheduling in NUMA systems, and follows with the implementation and evaluation of the proposed NUMA-aware thread scheduling and optimization in BBCP software. Chapter 6 presents the design, implementation and extensive evaluation of the new software system, named RAMSYS, followed by the conclusions in Chapter 7.

Chapter 2

Background

In the new era of the prevalence of multi-/many-core technology, the computation efficiency and performance are at an inflection point: on the one hand, the performance of serial applications and their underlying single-core servers saturates because of the power constraints and the problem of energy dissipation; on the other hand, a trivial migration of these applications to the multi-core and many-core platform does not guarantee any improvement, and might even cause unexpected aggravation. We continue observing a widening gap between the bare-metal performance of a multi-core processor and the effective application-level capacity while many vendor road maps promise to repeatedly double the number of cores per chip [26] based on the Moore's law. Furthermore, many hardware research/development efforts were to accommodate more nodes into a single system. In [27], the authors described that the coherent HyperTransport (cHT) protocol used in the AMD Magny-Cours Opteron processors cannot support more than 8 nodes. Although a new HNC HyperTransport specification was proposed to overcome this limitation, current AMD Opteron processors do not really implement this new protocol extension. In fact, the scalability of the current series of processors was improved by adding a new bridge chip which could then support up to 32 nodes effectively.

With the nodes in the system get bigger, many works illustrated a heavier NUMA asymmetric to I/O performance. The research [13] benchmarked TCP performance with the newly released 40 Gbps Ethernet technology on their two-node Intel testbed and eight-node AMD testbed. Their measurement showed that the placement of TCP processes on remote CPU cores, at either the sender or receiver side, can lead to as much as 30% loss of the overall TCP bandwidth performance. In [28], PCIe-attached GPU hardware was tested by various benchmark tools to compare the bandwidth performance of shared I/O hub and dedicated I/O hub in NUMA systems. The results showed that the penalty of incorrect NUMA assignment is substantial and asymmetric. A 31% reduction in read-back bandwidth and a 13% reduction in download bandwidth were observed for bulk data transfers to GPU devices. The study in [29] illustrated that remote transfers to SSDs could potentially reduce the maximum achievable throughput by 8% to 40%, and delay I/O completion time by up to 130%.

Except for NUMA-awareness, an effective multicore resource scheduler needs to explore various system parallelism while avoid the penalty of shared resource contention. Plenty of efforts have been made to propose proper resource scheduling mechanism, multi-threading and concurrent design for various applications. They provide good guidance for the research of high performance data replication systems. In the section, we summarize the background and related work to this dissertation.

2.1 NUMA Terminology

Advanced server technology can be leveraged to improve system performance and reduce overall cost [30–34]. Many current high-end systems include multiple CPU packages (sockets) on a single motherboard. Processor cores, memory banks, and Input/Output (I/O) modules are dis-

tributed across different domains (nodes) and assembled together by a cache-coherent, point-to-point interconnect. Each processor has faster access to directly-attached memory modules than to the remote ones that are linked by these interconnect buses. Such a feature is widely known as the Non-Uniform Memory Access (*NUMA*). As NUMA systems scale up, it is prohibitively difficult to implement a full connection for all processors in a host due to hardware constraints. For instance, the pin constraint of the AMD G34 (Generation 3, four memory channels) architecture allows at most four HyperTransport (HT) ports per CPU node. Additionally, for the bottom nodes in the topology, as shown in Figure 1-2, one port is reserved for I/O peripherals. These design constraints prohibit a fully-connected topology, and create different physical distances for remote accesses. In Figure 1-2, the two CPU dies in the ellipses are physically put in a tightly coupled multi-chip CPU package. We define "local" node as all the resources, including CPU cores, memory banks, I/O devices, directly attached to a single CPU die, and a "neighbor" node as the resources that are not local, but in the same CPU package, and "remote" nodes as the resources in other packages. For example, as shown in Figure 1-2(a), node 7 is local to itself, a neighbor to node 6, remote to nodes $\{0, 2, 4\}$ with one hop, and to nodes $\{1, 3, 5\}$ with two hops. The similar topology of the Intel eight-node NUMA architecture can be seen in [35], in which all NUMA nodes are also partially connected. This implies as well multiple possible performance levels to access data.

2.2 NUMA Characterization

The first step of utilize NUMA multicore system resource is to understand and characterize the NUMA effects to data replication performance. Previous NUMA characterization studies included extensive experiments to

evaluate the performance aspect of NUMA memory and peripherals access. In [36], a memory access cost model was built via the STREAM benchmark, and then confirmed with several other benchmark tools. In [28], the bandwidth performance of PCIe-attached GPU hardware was tested by various benchmark tools to compare two different design strategies for I/O hubs in a NUMA system. TCP performance with the newly released 40 Gbps Ethernet technology on two different NUMA architectures was benchmarked in [13]. However, all these efforts simply attributed the NUMA performance asymmetry to the spatial distance between nodes. There was no prior study on the concrete causes of NUMA remote access cost for contemporary high-performance hardware devices and their impacts on data replication applications.The author of literature [37] proposed a prototype software comprised of two transaction processing benchmarks and one OS customization. The data collected by the software was then applied to the task-dispatching and page-allocation algorithms of an OS, and led to a decrement of inter-node bus traffic in the NUMA system. Numerous performance models [38–40],were also extended to handle NUMA architectures, and can be utilized to determine the expected performance of a given application. However, these models required prior knowledge on memory-access latency and bandwidth in order to estimate the performance. In our performance model, we require little knowledge about a given system. In [41], a memory access cost model was built via the STREAM benchmark, and then confirmed with multiple benchmark tools. The same authors then integrated them into a benchmark toolkit called *cbench* [42]. However, in this work, we show that their methodology based on STREAM benchmark cannot be applied to the I/O performance model of modern HNC NUMA hosts. We also provide our own modeling benchmark to characterize the NUMA multicore systems.

Except for experimental benchmarking, hardware event counters were

also often utilized to analyze and improve performance over multicore systems. These counters can track thousands of events to monitor covering many aspects of microarchitectures behavior without slowing down the kernel or applications. Many tools to gather hardware performance counter data are available for Linux, including OProfile, likwid [43], PAPI [44], perf [45], pfmon [46], Intel VTune etc. The Clavis scheduler [47] is user level scheduler that supports various scheduling algorithms under Linux operating system running on multicore and NUMA machines. It uses hardware counters to monitor the system programs and predetermine the workload before scheduling. The work in [48] analyzed the impact of scheduling decisions on the dynamic performance of tasks. It utilized hardware event counts to track dynamic events at each core, then analyzed the data by core, by task, and by various thread schedulings and time slicings to show which counters mattered the most and which sorts of multicore schedules could lead to degraded performance. The dissertation also utilizes hardware event reading to compute NUMA scheduling factor, and thus supports dynamic resource scheduling on multicore platforms.

2.3 Mathematical Formulation of Multicore Scheduling

A number of studies focus on mathematically formulating the multi-task scheduling problem on multicore system [49–52]. They try to optimally schedule concurrent tasks to multiple cores in the system, such that the performance degradation is minimized, usually attained by minimizing the contention for shared resource while keeping good data locality. Unfortunately, even if all possible combinations of co-scheduled applications and their deleterious effects were known beforehand, optimal scheduling for

more than 2 cores on a processor is an NP-complete problem. The research in [50] targets at figuring out the optimal combinations of application process that lead to minimal shared resource contention. The problem is represented as a graph theoretic formulation whereby finding the minimum edge cover of a weighted graph. A polynomial-time algorithm for this formulation is then provided to find out the solution efficiently. However, it is only validated on dual-core and quad-core versions, and hard to scale to high-core-count systems. Another study [51] formulates the multicore scheduling problem using a Markov decision process (MDP) that considers power consumption of the processor cores and caches for video decoding applications. The MDP solution requires complexity that exponentially increases with both the number of processors and the number of frames in a short look-ahead window. The work in [52] addresses the problem of parallelizing and scheduling a set of sporadic parallel tasks. The authors formulate the problem as a density minimization problem with multiple parallelization options, and validate the proposed algorithm with both simulation and benchmark application. The issue is that it does not consider preemption and migration cost, and NUMA asymmetry. In this dissertation, we model the data replication task scheduling as min-sum-max resource allocation problem (MSMRAP), and show that it is a NP-complete problem.

2.4 Multicore-aware and contention-aware scheduler

There has been significant interest in the research community in addressing multicore-aware and contention-aware scheduling. We categorize them as follows.

1) **Hardware level**. These studies mostly fall into two categories,

performance-aware cache modification, e.g., [53–57], and performance-aware memory controller scheduling, e.g., [58–61]. The proposed solutions usually require changes to hardware and OS. As such, the majority of these techniques were only evaluated in simulation studies. Among them, some promising solutions have yet to be implemented in real product systems.

2) **OS level**. Thread scheduling policies at the OS level are grouped into two categories: thread-dependent policies, in which threads are scheduled based on their application types and relations to other threads, and thread-independent policies, where no application types and dependencies are considered in scheduling. In most general-purpose OSs, e.g., the default Completely Fair Scheduler (CFS) in Linux, use thread-independent policies and tend to schedule threads on less loaded cores for load balancing or cores with warm cache to exploit cache affinity. Thread-dependent schedulers appear mostly in research studies [62–65]. These mechanisms usually require intensive kernel modifications, and have yet to be deployed into real systems.

3) **User level**. Application-level schedulers [66–68] to address thread-to-resource mappings are particularly attractive because they require no change to hardware and only minimal modification to the operating systems. Most modern OSs, such as Windows, Linux, and Solaris, already provide a rich collection of application programming interfaces (APIs) to enable application-level NUMA optimization [69].

This dissertation focuses on user level benchmark tools and scheduler.

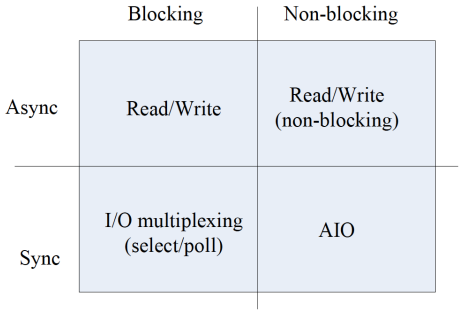


Figure 2-1: I/O modes in Linux

2.5 Parallelism and Concurrency at Various Levels

Except for NUMA-awareness, highly efficient data replication systems are also required to exploit the parallelism and concurrency that are available at various system levels. In this section, We discuss a taxonomy of four operation models [70] and parallelism levels, with the combinations of synchronous/asynchronous I/O and blocking/non-blocking models as show in Figure 2-1.

- **Synchronous blocking I/O.** The synchronous blocking I/O model is the most common model to a user-space application that performs a system call and blocks itself until the system call is complete. It is simple to implement and efficient in term of CPU cost for the single process and single core paradigm. To improve parallelism, programmers must utilize multiple threads to keep multiple I/O requests in-flight. We term it "thread-level parallelism. The overhead of keeping multiple active threads becomes a disadvantage. For example, we run iperf tool [25] over our 40 Gbps Ethernet testbed described in Section 6.3.1. Iperf test reaches 30 Gbps bandwidth with single thread, peaks at 40 Gbps (the bandwidth limit) with 8 parallel threads, and nevertheless deteriorates to only 12.5 Gbps when 500 iperf threads

send traffic.

- **Synchronous non-blocking I/O.** A less efficient variant of synchronous blocking is synchronous non-blocking I/O. Regardless whether the actual I/O completes, an I/O call/request returns right away, either successfully or with an error code indicating that the command could not be immediately satisfied. This may require an application to make numerous calls, and triggers numerous context switches, to busy-wait for the final completion of data retrieval. This model is potentially extremely inefficient because any gap between the data becoming available in the kernel and the user post an I/O request to return it can increase the processing latency.

- **Asynchronous blocking I/O.** Another blocking paradigm is that an application makes non-blocking I/O calls and uses blocking calls afterward to check the status, i.e., fetch notifications. In this model, an application uses two blocking systems calls, i.e., pollings (select() or poll()), to determine and detect any new activity/event associated with an I/O descriptor. To improve parallelism in this paradigm, application can utilize a polling system that provides notifications for multiple descriptors simultaneously. We term this as "descriptor-level parallelism. The application blocks on polling notification instead of I/O requests. GridFTP tool adapts this model and uses a single polling thread to manage both disk I/O and network I/O descriptors. The primary issue with this mode is that the polling thread usually incurs non-negligible overheads, and becomes bottleneck in the high performance I/O scenarios.

- **Asynchronous non-blocking I/O.** Finally, the asynchronous non-blocking I/O model overlaps I/O activities and other tasks (e.g. computing) within a single thread. An I/O call/request returns imme-

diately, indicating that OS successfully initiates I/O activity. The application program then performs other tasks, such as posting more I/O requests or handling the completion notification of previous requests, while the background I/O operation proceeds concurrently. We refer this method as "block-level parallelism. Linux system has two sets of Asynchronous non-blocking I/O (AIO) implementations. One is POSIX AIO, an application-level implementation that essentially emulates the behavior of asynchronous I/O with multiple background threads performing regular blocking I/Os, and hence gives the calling party an illusion that I/Os are asynchronous. The other one is Linux native AIO [71] directly supported in the OS kernel level.

In this paper, we only consider the multi-threaded synchronous I/O model and the asynchronous non-blocking I/O to parallelize storage I/Os given the apparent shortcomings of the other two modes.

2.6 Existing Multicore-aware Data Replication Software

There are also extensive works to adapt end-to-end data replication to the multicore systems in Linux. Scp and FTP tools are known to be single-threaded. Herein, we show the multi-threaded high-speed data replication tools as follows.

2.6.1 TCP-based tools

GridFTP defines a general-purpose mechanism for secure, reliable, high-performance data movement, and it is the most popular tool for high performance data replications. It was first reported in the research [19]. Then, the authors in [72] showed performance limitations of GridFTP through several

numerical examples on wide-area area environment. GridFTP-APT proposed in [73,74] could automatic tune the GridFTP parallelism to improve the performance of GridFTP in various network environments. Based on GridFTP, the work in [75] an algorithm that dynamically schedules a batch of wide-area data transfer requests with the goal of minimizing the overall transfer time. Recent studies with GridFTP can be found in [76–78]. dd

The BaBar Copy Program (BBCP) is another widely-used peer-to-peer data replication tool over TCP/IP stack. It was first described in [79]. Its excellent performance has been identified in multiple studies [80–82]. The Fast Data Transfer service (FDT) [83] is also an high-speed data replication tool using multiple parallel TCP streams.

2.6.2 UDP-based tools

UDP-based data transfer (UDT) [84] is a reliable UDP based application level data transport protocol for distributed data intensive applications over wide area high-speed networks. UDT uses UDP to transfer bulk data with its own reliability control and congestion control mechanisms. GridFTP also extends a separate module to integrate UDT in [85]. Aspera is another UDP-based solution as an industrial product. It relies on FASP protocol [86] to deliver high performance over the Internet. The work in [87] proposed Performance Adaptive UDP (PA-UDP) protocol to dynamically and autonomously maximize performance under different systems. These tools are usually designed for 1/10 Gbps network, and will lead to prohibitive system load while the network performance reaches 40 Gbps or higher.

2.6.3 RDMA-based tools

Remote Direct Memory Access (RDMA) is introduced to eliminate data copy overhead in high performance networks [88]. A number of protocol were proposed to take advantage of this technology for both local area and wide area data replications. For example, the study in [89] provided the Advanced Data Transfer Service (ADTS) with RDMA send/receive operations, and the authors in [90] integrated this service with GridFTP. Another research [22] gave RDMA-based FTP (RFTP) tool using RDMA_WRITE operation, and demonstrated its outstanding performance over various real-world network testbeds. However, RDMA-based tools need special network switches, adapters and cables. The cost of these hardware is usually high.

Chapter 3

NUMA effects Analysis and Quantification

The first step to design resource scheduling mechanism for high performance data replication is to understand the essence of NUMA effects, and hardware event counters are powerful tools to discover the interaction among underlying hardware components. In this chapter, we first demonstrate the huge performance gap between hardware capability and actual I/O operations using a state-of-the art NUMA system and widely adopted benchmark tools. An in-depth analysis, using hardware event counters, then reveals the underlying causes of the gap, and justifies the need for NUMA-awareness for modern data replication applications to eliminate this gap. Based on the new findings, a new metric is proposed to quantify the real-time NUMA penalty in multicore systems.

3.1 Experimental setup

We focus our experimental study on a Dell PowerEdge R820 host. Table 3.1 summarizes its hardware and OS configurations. This host supports 64 bit PCI-Gen3 peripheral interfaces, each of which can potentially provide

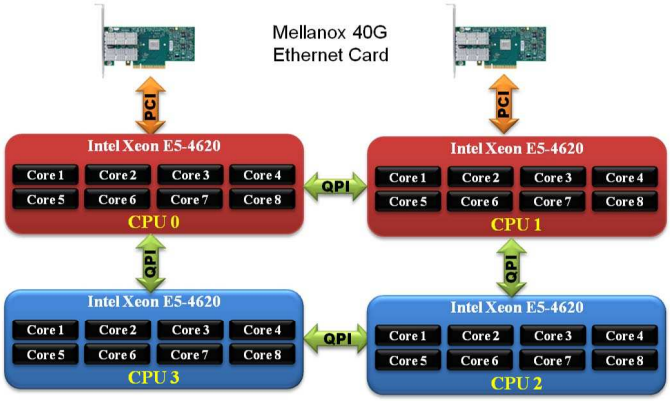


Figure 3-1: System topology of evaluation system

64 Gbps bandwidth. The evaluation system contains 32 CPU cores. The cores are grouped into four CPU packages, a.k.a. CPU nodes. Each NUMA node herein consists of eight CPU cores, a shared last level cache (LLC), one memory controller, and one I/O subsystem. QPI interconnect paths enable communication among nodes. The system topology is shown in Figure 3-1. The four NUMA nodes are arranged with a ring topology. Only node 0 and node 1 have on-chip I/O hubs or PCI bridges. Node 2 and 3 are connected to the peripheral devices only via Node 0 and 1. Two parallel QPI channels are deployed between neighboring nodes. *hyperthreading*, *irqbalance*, and *cpuspeed* services are disabled during all tests based on the recommended settings in [91]. All these hardware setups guarantee a high aggregate memory and network performance.

Figure 3-2 shows the network connectivity of the testbed system. The system is configured with two 40 Gbps Ethernet adapters. Two back-to-back Ethernet connections are established between the two tested nodes with optical fiber cables. The other end of cables is connected to an IBM test host. All network adapters are PCI Gen3 based. The evaluation system is also connected to a storage system via 40 Gbps InfiniBand (IB) [92] interface. This backend storage-area network (SAN) in the figure is based on the iSCSI extensions for RDMA (iSER) protocol [93], and can sup-

Table 3.1: Server specifications

Processor model	Intel Xeon CPU E5-4620 @ 2.20GHz
CPU microarchitecture	Intel Sandy Bridge EP
CPU cores	4×8
LLC size	16 Mbytes
System memory	4×192GB DDR3 1600MHz
Board chipset	Intel C600
QPI bandwidth	2×7.2 GT/s (115.2Gbyte/s)
Operating system	CentOS release 6.3 (Final)
Linux Kernel	2.6.32-279.5.2.el6.x86_64

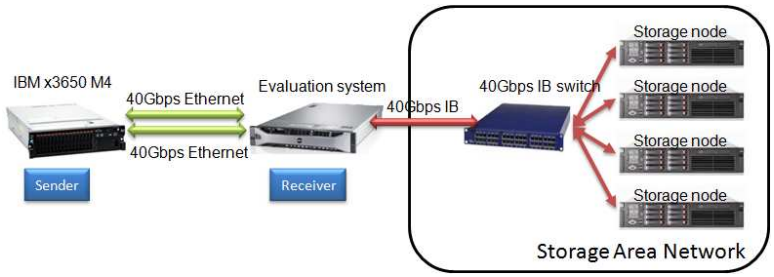


Figure 3-2: System connectivity for network performance characterization

port a maximum read/write bandwidth of 40 Gbps. We also bind network interrupts in the evaluation host to the directly attached CPU nodes, respectively. This is the optimal configuration for interrupt affinity, as stated in [91]. Table 3.2 gives other network parameters.

Table 3.2: Network specifications

Network adapter	Dual-port Mellanox ConnectX-3 EN 40 Gigabit Ethernet adapters
Network driver	MLNX_OFED_LINUX-1.5.3-3.1.0
Round trip time	0.06ms
MTU	9000 bytes (Jumbo Frame)

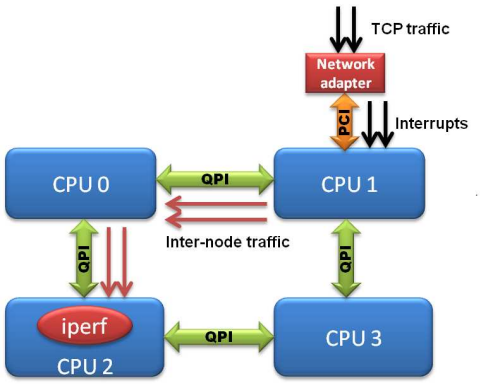


Figure 3-3: Remote network adapter access by iperf benchmark

3.2 Observation of NUMA Effects on Network Performance

Here we use *iperf 2.0.5*, a widely used bandwidth benchmarking tool, to determine the network bandwidth performance of the evaluation system. Since iperf mimics the behavior of typical bulk data replication applications, such as intensive memory data access and protocol stack processing, and it is a proper tool to discover the NUMA effects on data replications.

In these iperf tests, we found that the location of memory node did not show noticeable impacts on performance. This is because the operating block size in the iperf tests is relatively small, and this enables iperf to take advantage of high-speed caches between CPU and main memory. However, the network card in this test is physically attached to node 1, and all the interrupts from the card are also directed to this node. For an example in Figure 3-3, if we pin the iperf process to node 3, it needs to retrieve data content from the interrupt handler at node 1. Hence, TCP applications suffer from the NUMA remote access penalty as well.

Furthermore, the location of the iperf sender node has minor influence to the data transfer performance because the bottleneck of TCP data transfer resides in the receiver side, and has direct impact to the end-to-end network

Table 3.3: Iperf benchmark test parameters

TCP congestion control	Cubic
TCP window size(Kbytes)	128
I/O block size(Kbytes)	128
Test duration(seconds)	300

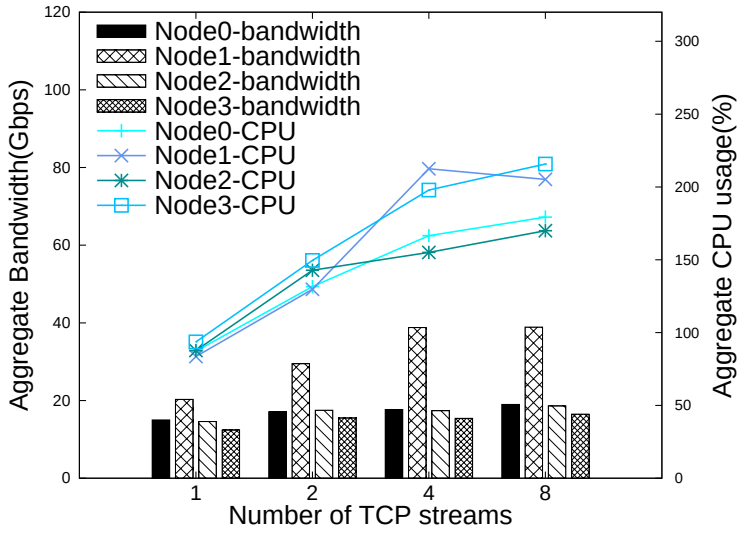


Figure 3-4: Aggregate bandwidth (higher is better) and CPU usage (lower is better) of iperf benchmark with multiple concurrent streams, "Node n" represents that all the test processes are running on node n.

performance. This observation is also confirmed in [13]. We focus on the receiver behavior and deploy the evaluation system as the receiver, and the remaining IBM host as the sender, as shown in Figure 3-2. Other iperf test parameters are listed in Table 3.3.

The iperf receiver processes are pinned into each of the four CPU nodes by using *numactl* utility [94]. Each test employs a different number of concurrent TCP streams. Figure 3-4 illustrates the aggregate bandwidth and CPU usage of each test case. We observe that the location of the CPU that hosts the benchmark processes has a remarkable impact to the network performance. NUMA nodes (node 0 and node 2) that are one-hop away have significantly lower bandwidth than the local memory node, and the two-hop node (node 3) has the worst memory bandwidth. As expected, this is

consistent with the system topology. What is not expected is the enormous NUMA penalty to remote network adapter accesses. With the local binding configuration, iperf can easily reach the physical bandwidth limitation (40 Gbps). However, although the evaluated system supports a QPI bus bandwidth of 115.2 Gbyte/s and a memory bandwidth of 51.2 Gbyte/s, the bandwidth of the remote binding cases cannot even exceed 20 Gbps. Furthermore, each hop on the QPI bus does not imply the same performance penalty to network I/O performance, i.e., one-hop performance is only a half of the local case, but the two-hop case shows only a slight degradation, or less than 15%, compared with the one-hop case. The bandwidth of local access improves significantly with the growth of the number of parallel threads, while that of remote memory access almost stays constant due to the NUMA penalty.

3.3 Analysis for NUMA Remote Access Penalty

In the previous subsection, we illustrated that the performance seen by data replication applications would be much lower than the maximum hardware capability when remote access was involved. This motivates us to consider, can the further hardware performance improvement eliminate the NUMA effects on these applications? Will the NUMA-awareness still be necessary to future data-intensive applications, including bulk data replication tools? The answer to these questions entails an in-depth understanding of the factors lead to NUMA effects, instead of a simple attribution to the physical distance between nodes.

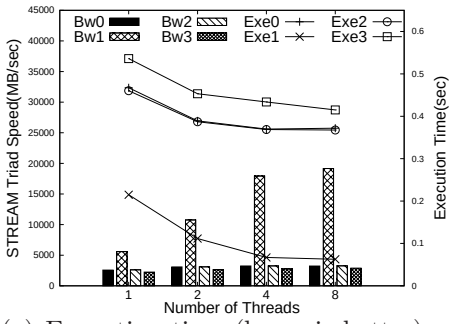
For these purposes, we use the STREAM memory benchmark for analysis instead of iperf network benchmark due to following reasons. Firstly, TCP-based operations in iperf benchmark impose intensive memory accesses, and this is very similar to the operations in the STREAM bench-

mark. Secondly, network I/O bandwidth is often an order of magnitude lower than memory bandwidth, and thereby the I/O performance difference among various NUMA configurations is not as obvious as the memory access cases, as detailed in [95]. Finally, the operations in the STREAM benchmark are easy to understand, and thus facilitate our comparison and analysis.

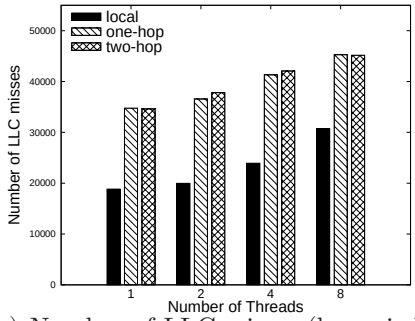
3.3.1 Observation of NUMA effects on memory benchmark

In order to obtain an accurate memory performance, STREAM requires that each data array be four times as big as the largest cache used. In our case, the LLC is 16 Mbytes per CPU socket, and thereby the array size should be at least 64 Mbytes, or $N = 8,000,000$ double-precision floating point numbers. For our tests, we set $N = 60,000,000$. Also, we change the number of iterations from 10 to 100, for a more accurate performance measurement. It is not necessary to try all the combinations of CPU-node binding and memory-node binding since in the evaluated testbed, performance levels are only determined by the number of intermediate physical links (i.e. hops) on the path between two NUMA nodes. Therefore, we pin all the benchmark threads on node 1, to keep consistency with the former network data replication tests, and test the performance of node 1 to allocate and access data in memory banks that attached to all four nodes. The performance of the STREAM *Triad* operation is shown in Figure 3-5(a). "MEMn" denotes the case when the test threads are running on node 1, and access data on node n. "Bwn" represents the bandwidth performance when accessing data on node n, and "Exe$n$" is the corresponding execution time. Other STREAM operations produce similar results.

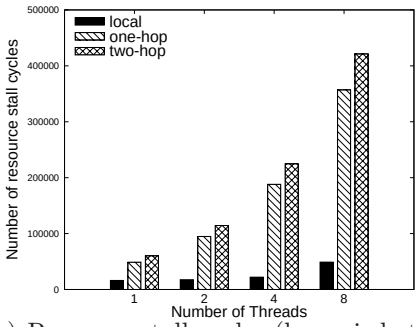
As shown in Figure 3-5(a), the performance characteristics given by the



(a) Execution time (lower is better) and bandwidth (higher is better)



(b) Number of LLC misses (lower is better)



(c) Resource stall cycles (lower is better)

Figure 3-5: Primary performance analysis for STREAM Triad operation with multiple concurrent threads

STREAM benchmark are close to that shown by the iperf benchmark. As expected, the NUMA effects in STREAM results are more observable than that of iperf results. For example, in the case with eight concurrent test threads, the bandwidth of one-hop memory access drops as much as 83% compared to local accesses. Single local thread cannot reach the maximum memory bandwidth because of the constraint on the total number of outstanding LLC misses per thread, as shown in [96].

3.3.2 Penalty indicated by LLC misses and memory access stalls

To get the insight into hardware behavior during the experiments, the *OProfile 0.9.7* [97] software is utilized to monitor various hardware performance counters. This software allows us to determine the number of

events that occur during a test run. These events are recorded by various processor event counter registers, and collected by the OProfile tool. When a profile is created over a long test run, it provides accurate counts to the tracked events for comparison, given that the sampling rate is configured appropriately. However, there is a huge set of hardware event counters, and the readings for each counter also have different meanings over different platforms. Herein, we carefully inspect these event counters, and focus on the ones that are relevant to system architecture, have primary impact to the memory and network performance, and also demonstrate noticeably different patterns across different test scenarios.

The first metric is LLC misses. The LLC miss rate is a common indicator of performance penalty used by many multicore related studies [66, 98]. Any memory request that experiences LLC miss must be served by local or remote memory, and thereby will trigger more memory resource stall cycles and decrease CPU utilization. If it takes two clock cycles to move data between two adjacent components inside a processor chip, the latency for a memory access will be roughly $2(N + 4)$ cycles, according to study [99], where N is the number of cores. When $N = 8$, this latency is equivalent to the penalty of 24 cycles or nearly 10 ns, and a remote memory access will further amplify this latency, depending on the NUMA property of individual system. Hence, the number of LLC misses provides important information to indicate the overall memory access performance. Figure 3-5(b) shows the total number of LLC misses during the previous STREAM benchmark tests. Hereafter, all the test threads are running on node 1. "local" denotes the case of accessing data affiliated with node 1. "1-hop" represents the case of accessing data on node 0 and 2, and "2-hop" is the case of accessing data on node 3. To move the same amount of data in the STREAM benchmark, the remote access experiences 47% to 84% more LLC misses than the local access among various test cases. This im-

plies that a remote access incurs more costly memory operations, and thus leads to a greater performance penalty.

However, LLC misses served by remote DRAM require even more cycles than the ones served by a local DRAM. For example, as mentioned in research [100], LLC misses answered from a local DRAM cost about 180 cycles, while those served by a remote DRAM need about 300 cycles. This prompts us to monitor the RESOURCE_STALL counter which tracks the number of CPU idle cycles during resource-related stalls. Instead of simply measuring the number of long-latency events, like cache misses, this counter measures the actual performance penalty due to long-latency events. It is an iron-law measurement of various performance degradations in the underlying circuitry. In Figure 3-5(c), we present the number of resource stall cycles during the STREAM tests. The combination of Figure 3-5(a) and Figure 3-5(c) confirms that the increment in CPU stall cycles correlates well with the actual penalty to memory access bandwidth. During these resource stall cycles, CPU can do nothing but wait for its data. In the case of remote accesses, a great amount of CPU time is wasted on the resource stall cycles incurred by the usage of inter-node bus. Meanwhile, during the STREAM tests, *nmon* tool [101] is employed to monitor system resource utilization, and we observe that one benchmark thread always completely occupies one CPU core with a 100% busy status. Since in every test case the system executes the same operations with the same workload, and each test thread requires the same CPU usage, a significantly higher bandwidth of local memory access implies a much more efficient use of CPU cycles.

The experiment result also shows that the CPU stall cycles in the remote memory access tests increase almost linearly with the number of concurrent threads, while those of the local memory access tests stay relatively constant. In other words, while data travels through inter-node bus, every STREAM thread is frequently blocked and waits for data to be transferred

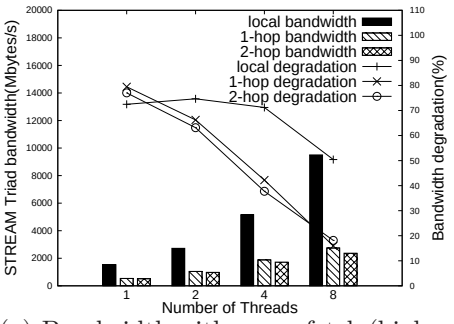
from remote memory. This suggests a bottleneck occurs at the inter-node bus.

3.3.3 Underlying reasons of NUMA Penalty

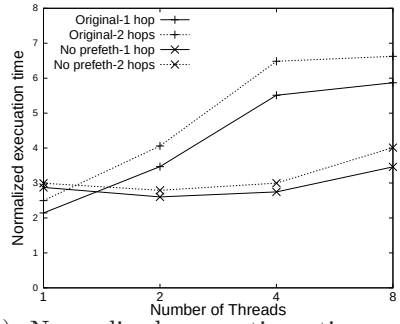
Resource stall cycles give us an overview of performance overhead, but they do not explain why this difference exists. In this part, we provide an in-depth analysis on the detailed causes of the NUMA remote access penalty that may be imposed on bulk data replication.

Prefetch contention. Contention for resources involved in prefetch processes begins to draw attention in recent research studies, e.g. [66]. Herein, prefetching resources may involve all hardware resources that could affect the efficiency and effectiveness of prefetching, including memory controller, QPI bus and instruction/data cache. It is difficult to break down the performance degradation due to each of them. Therefore, we compare the performance variance before and after all hardware prefetchers are turned off. Figure 3-6(a) shows the STREAM bandwidth measurements with no hardware prefetcher and their percentile degradation compared to the previous tests with prefetchers. In general, disabling prefetchers leads to performance penalty in all test cases, in terms of more LLC misses, more resources stall cycles, and lower bandwidth. This can be ascribed to the fact that prefetchers are quite effective to the sequential memory access operations in the STREAM benchmark. However, a local access experience a larger performance degradation if prefetch is turned off than a remote one when there are multiple concurrent threads. In other words, in the presence of prefetchers, a local access experiences much less prefetch contention among task threads, and thus can make more efficient utilization of prefetch resources.

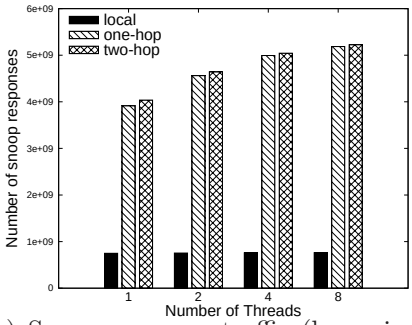
In order to analyze the variation in the NUMA performance asymmetry, we calculate the normalized execution time of the STREAM Triad



(a) Bandwidth with no prefetch (higher is better), and bandwidth degradation after turning off all the hardware prefetchers (lower is better)



(b) Normalized execution time with-/without prefetch (lower is better)



(c) Snoop response traffic (lower is better)

Figure 3-6: Prefetch and snoop traffic characterization of STREAM Triad operation with multiple concurrent threads

operation as Equation 3.1, and show the result in Figure 3-6(b). "Original" denotes the cases that all hardware prefetchers are enabled, and "no prefetch" means all of them are disabled. This normalized execution time is the inverse of the normalized bandwidth. The larger the normalized execution time, the more asymmetric the performance is. When there is a single test thread, the similar asymmetry is observed in both prefetch and non-prefetch cases. However, with more than one test threads in the presence of prefetchers, the contention on prefetching resources starts to incur more cost on the remote memory access, and results in a greater performance asymmetry. After prefetchers are disabled, the system become less asymmetric, and this asymmetry stays constant even with many more

test threads. One possible explanation is that remote prefetches cost more time, and are more likely to conflict with each other. This would lead to a lower prefetch efficiency when multiple competing threads exist.

$$\text{Normalized execution time} = \frac{\text{Execution time in remote case}}{\text{Execution time in local case}} \tag{3.1}$$

Cache coherence traffic. To explain the performance asymmetry, we also look into cache coherence traffic. Modern multiprocessors adopt hardware-based cache coherence protocols to ensure that all changes in distributed caches are propagated throughout the entire system in a timely manner. The interconnect between processors in our study, Intel QPI 1.1, also supports a home snooping cache coherence protocol. The detailed description for this protocol can be found in [102]. When a core encounters cache misses in the LLC, it must snoop the other CPU sockets to check their caches. In the QPI architecture with the home snooping cache coherence protocol, a remote snoop requires up to four steps and generates a significant amount of inter-processor QPI traffic. Although the size of each snoop request/response is small, the frequency of snoops can be very high. It occurs for every cache miss, sometimes even for cache write hits as well. The QPI bus between processors is shared by the snoop traffic and actual data traffic. In the case of remote cache and memory access, snoop communication frequently holds the bus for cache coherency use. This greatly increases the bus turnovers and interruptions to actual data replications, and creates an inter-node bus bottleneck to large data flows.

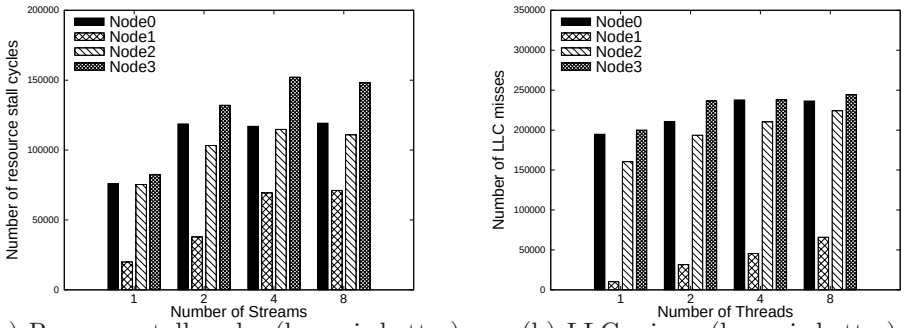
On the other hand, for local memory and cache accesses, all processors implement a snoop filter [103] that reduces cache coherence traffic by filtering out local requests because local requests can then be served locally

without involving the QPI bus. With this type of filters, a CPU core that encounters cache misses only needs to synchronize with other cores in the same CPU package, and thereby the synchronization cost would be greatly reduced. Sandy Bridge microarchitecture allows for measuring the snoop traffic using the OFF_CORE_RESPONSE counter together with the snoop response filter. The detailed method is covered in [104]. Figure 3-6(c) illustrates the snoop response counts in both local and remote cases with all prefetchers. One-hop remote access generates 485% to 673% more snoop traffic than a local access, and two-hop one incurs 487% to 677% more traffic in various test cases. The snoop traffic recorded without all the prefetchers also reflects the similar results. This implies that the constant snoop traffic penalty is the leading cause of the consistent performance asymmetry described in the non-prefetch cases of Figure 3-6(b).

Herein, our observations and analysis are also quantitatively consistent with vendor's document. In Dell's public document [105], the authors also utilized the STREAM benchmark to characterize the memory bandwidth of a Dell PowerEdge R720 server equipped with dual Intel Xeon E5 processors that are similar to the ones used in our study. This host also has the same type of QPI as our evaluation system. As they reported, the NUMA remote memory access (one-hop) bandwidth is reduced by 55% compared to local access bandwidth. With the number of NUMA nodes increases to four, as in our evaluation system, the synchronization cost would increase as well, and cause a performance degradation of 82.5% for one-hop remote memory access.

3.4 NUMA scheduling factor

In order to confirm the iperf network benchmark shares the same NUMA effects as described earlier, the hardware counter characterization of iperf



(a) Resource stall cycles (lower is better) (b) LLC misses (lower is better)

Figure 3-7: Hardware counter characterization of iperf benchmark with multiple concurrent streams

benchmark is shown in Figure 3-7. As expected, the local access to network cards introduces much fewer CPU stalls and LLC misses, and thereby avoids the aforementioned multiple NUMA remote access penalties. In addition, PCI bridge is integrated into CPU chip in our evaluation system. The fast on-chip data path further boosts the local access performance. The prefetch and snoop characterizations are not included here because they are quite similar to that of the STREAM results.

In summary, we conclude that the NUMA remote access penalty is significantly pronounced by the joint effect of hardware prefetch contention and CPU synchronization cost. These bottlenecks have not been eliminated even the bare-metal hardware performance keeps increasing, and thereby, NUMA-awareness is still vital for data replications to take the full advantage of hardware capability now and in the future.

Based on the findings in this section, we proposed a new NUMA scheduling factor as the Equation 3.2. At first, based on the research [14], a bigger link width would decrease the NUMA effects. On the other hand, a higher cache coherency traffic, i.e. snoop traffic, and a longer prefetch latency imply a greater NUMA penalty. Therefore, the proposed NUMA scheduling factor quantify the degree of this penalty of remote resource access. This factor provides a good guidance to the system scheduler to decide the best

node when it needs to migrate a thread to remote resource in real time.

$$\text{NUMA factor} = \frac{\text{Cache coherency traffics} \times \text{Prefetch latency}}{\text{Link width}} \tag{3.2}$$

3.5 Summary

As high-performance networks, such as 40/100 Gbps Ethernets, emerge as an integral part of large-scale data-intensive computing systems, today's data replication tools cannot fully utilize the maximum capacity of modern NUMA hosts. In this chapter, we characterized the NUMA access by bulk data replication applications on a state-of-the art host. The experimental results revealed the penalty of accessing remote resources, due to more prefetch contention, higher synchronization cost, and a larger amount of cache coherence traffic. This was not quantitatively evaluated by previous studies. We thus concluded that the recent hardware performance improvement does not eliminate the NUMA bottleneck, and NUMA-awareness is still critical to bulk data replications. A new metric, NUMA scheduling factor is also proposed to support online thread/data mapping in multicore system.

Chapter 4

NUMA I/O Performance Modeling

Previous Chapter 3 proposes the NUMA scheduling factor to enable real-time NUMA-awareness. The advantages of this approach are the adaptivity to dynamic system load and accurate NUMA penalty quantification to pick the right node for thread/data migration. However, it also introduces several shortcomings. 1) Difficulty to get cache coherency traffic and prefetch readings online. Different platforms require different hardware event counters and filters to obtain those readings. Some servers may even do not support these events in CPU monitor modules. User may need to customize the application for different systems. 2) Overhead of dynamic scheduling. For each thread/data migration, the scheduler needs to get event readings for all NUMA nodes, computes the scheduling factors, and then make decision. Especially, users often need to transfer small files, where each data replication only requires a short time. The scheduling overhead is unneglectable. The problem becomes even worse in multi-user and multi-task scenarios, and frequent schedulings and migrations occur in the system.

The chapter first proves the ineffectiveness of using STREAM bench-

mark to build I/O performance model, and then designs and implements a new benchmark tool. At last, we evaluate the new tool, and identify its multiple usages in different user cases. With the new proposed tool, the users can first generate the performance model. The system scheduler then use this pre-defined model for thread/data mapping and migration. This avoids repetitively system profiling and computation while ensuring good NUMA affinity.

4.1 System Configurations for Characterization

In this section, we describe the configurations of the testbed hardware and the benchmark software in our experiments.

4.1.1 Server hardware specifications

Table 4.1 lists the system models and configurations. Both servers have four CPU packages, but in the AMD platform, each package contains two CPU dies, and thereby two NUMA nodes. The possible CPU topology of the eight-node AMD server can be any one shown in Figure 1-2. The Intel platform has a simpler architecture, and we used it in the previous Chapter 3. Its four NUMA nodes are arranged in a ring topology. Only node 0 and node 1 have on-chip I/O hubs or PCI bridges. Both our AMD and Intel hosts are equipped with two LSI Nytro WarpDrive WLP4-200 SSDs and a 40 Gbps network adapter with the capability of RDMA over Converged Ethernet (RoCE) [106]. Each target host is connected to another server to perform end-to-end network performance tests, as shown in Figure 4-1(a) and Figure 3-1. All network adapters and SSDs are directly attached to node 7 in the AMD testbed and node 1 in the Intel testbed, and these

Table 4.1: Server specifications

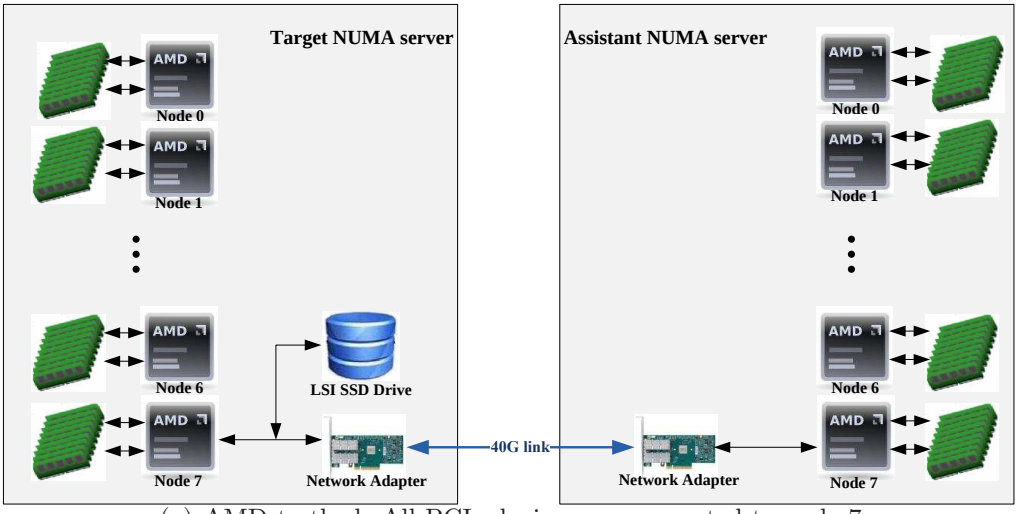
Motherboard	HP ProLiant DL585	Dell PowerEdge R820
Processor model	AMD Opteron 6136 Magny-Cours @ 2.4GHz	Intel Xeon CPU E5-4620 @ 2.20GHz
CPU cores/NUMA nodes	32/8	32/4
Last-level cache size	5Mbytes	16 Mbytes
System memory	32Gbytes DDR3	768Gbytes DDR3
Board chipset	AMD SR5690	Intel C600
Interconnect protocol	Hypertransport 3.0	QPI 1.1
Interconnect bandwidth	4.8 GT/s (9.6 Gbyte/s)	7.2 GT/s (14.4 Gbytes/s)
I/O Bus	PCIe Gen2 x8 lanes	PCIe Gen3 x8 lanes
Operating system	CentOS 6.3	CentOS 6.3
Linux Kernel	2.6.32.el6.x86_64	2.6.32.el6.x86_64

nodes will be used as the exemplary CPU nodes hereafter in their testbed servers respectively, as depicted in Figure 4-1.

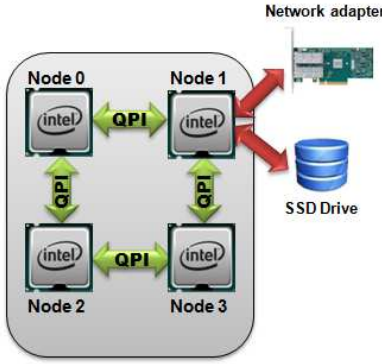
All network interrupts of the two platforms are delivered to the directly attached CPU nodes (exemplary nodes). All the network interface are set to support Jumbo Frame, and Round Trip Time (RTT) of all the network links is about 0.06 ms. TCP *Cubic*, a TCP variant is used here. *Hyperthreading*, *irqbalance*, and *cpuspeed* services are disabled during all tests for simplicity based on the recommended settings in [91].

4.1.2 Benchmarks and affinity settings

The focus of this chapter is to model the relative user-level bandwidth performance under all possible NUMA scenarios. As stated in [95], improving the absolute bandwidth can lead to more noticeable NUMA effect, and thus facilitate our comparison and analysis. Therefore, we optimize our testbed and benchmark settings as follows.



(a) AMD testbed. All PCIe devices are connected to node 7.



(b) Intel testbed. All PCIe devices are connected to node 1.

Figure 4-1: System connection diagram

Memory benchmark

The STREAM benchmark is used to determine the maximum aggregate memory bandwidth between NUMA nodes. The performance measurements reported by STREAM highly depend on compilers. We optimize the benchmark compilation according to the recommendation of AMD technical document [107], and also take advantage of the *-fopenmp* compiling flag to enable its multi-threaded capability. The STREAM benchmark executes four types of memory access operations on large data arrays, but they exhibit similar performance on modern machines. Herein, we choose

the *Triad* operation for our characterization, to keep consistent with tests in previous chapter. In order to eliminate the CPU cache affect, STREAM requires that each array be at least four times the largest cache used. In our case, the largest LLC size here is $16Mbytes$ per Intel CPU die, and thereby the array contains at least $64Mbytes$, or $8,388,608$ long integers.

I/O benchmark

Flexible I/O Tester (fio) [108] is the user-level benchmark tool used to characterize system's PCIe device access performance. This test tool spawns a number of processes performing I/O jobs with user-defined I/O operations and desired parameters. It supports a wide spectrum of I/O operations, such as disk read/write and TCP/UDP data transfer. We also add RDMA engines into this tool, and extend its capability to support RDMA_READ, RDMA_WRITE, and SEND/RECEIVE operations [22]. Furthermore, we redirect hardware interrupts generated by I/O devices to their local CPU node.

Even though each type of data accesses aforementioned uses a different set of resources during the data replication process, they all suffer performance penalty from untuned resource assignment. One main objective of this dissertation is to discover the patterns among these accesses, and extract a generic I/O bandwidth performance model from it.

4.2 Experimental characterization

4.2.1 Memory performance characterization

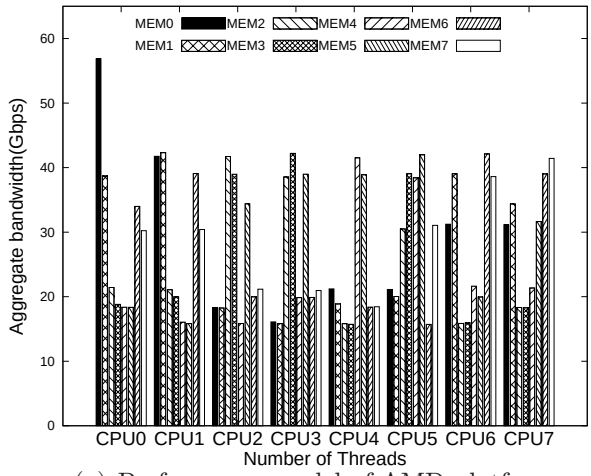
In this section, for each test case, the STREAM benchmark is set to run 100 times, and it reports the maximum observed bandwidth instead of the mean value. We use *numactl* to pin each benchmark process/thread to a desired

CPU/memory node. Benchmarking every memory data access scenario in a NUMA system entails the mapping of benchmark threads and their memory in every possible configuration that an application process might encounter during its execution. For modern multi-core systems, mapping threads to individual cores greatly expands the size of testing set. Herein, we assume that cores attached to the same NUMA node show identical memory and I/O bandwidth when accessing data on a given node. This assumption is appropriate in most scenarios as shown in previous literature [13, 41]. Hence, we need only focus on node-level characterization. Meanwhile, the number of parallel test threads is set to be the number of CPU cores in each NUMA node, which is four for the AMD testbed and eight for the Intel testbed. After benchmarking, a $N \times N$ bandwidth matrix can be obtained, where N is the number of configured NUMA nodes in a host. This is shown in Figure 4-3. We describe our observations of the bandwidth matrix as follows.

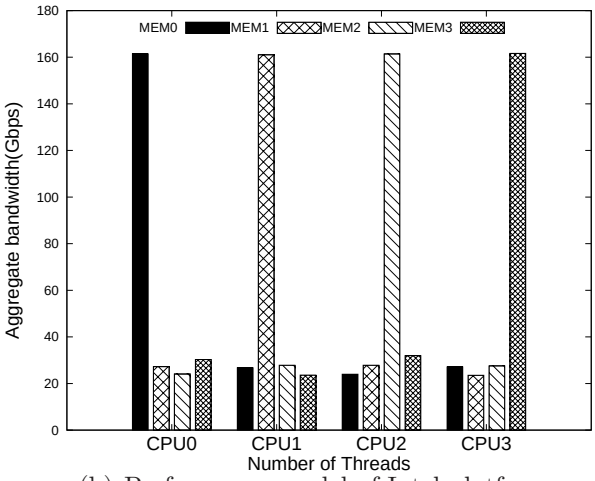
Observation 1. *Ineffectiveness of hop-distance in the AMD evaluation system.*

The bandwidth performance of the AMD testbed in Figure 4-2(a) exhibits asymmetry among tests. For example, when STREAM runs on Node 7 to access data from Node 4, we obtain a bandwidth of 21.34 Gbps, which is better than the two cases of accessing data residing in Node $2, 3$ respectively. However, when the benchmark runs on Node 4 to access data in Node 7, only 18.45 Gbps is achieved, and this is worse than the performance of running the benchmark tool on Node $2, 3$. This issue can result from the asymmetric setup in the AMD hardware, such as the number of request and response buffers, the configuration of link width for cache coherent traffic, and the routing mechanism in the tested topology [109, 110].

Furthermore, from the data in Figure 4-3, we can see the local node



(a) Performance model of AMD platform



(b) Performance model of Intel platform

Figure 4-2: Memory bandwidth performance model by STREAM benchmark Copy operation. "CPUn" denotes all STREAM test threads running on node n, and "MEMn" denotes all test STREAM threads are accessing the data in Node n.

has the best performance, and the neighboring node has the second best. The neighboring node is one hop away from local node, but it has better performance than all other nodes that are one hop away. That is because it has on-chip access to local node, and the link width between neighboring nodes is 24 in total, which is 1.5x-3x bigger that other nodes. Thus it performs better than other off-chip remote nodes. If we use hop-distance as the decisive factor of the NUMA cost, then we should have observed: 1)

Table 4.2: Bandwidth NUMA factor of of the Intel platform

Node ID	0	1	2	3
NUMA factor	5.94	1	5.89	6.63

The nodes with the highest bandwidth are local nodes. 2) The second best ones are the nodes that are one hop away. 3) The nodes with two hops always have the lowest bandwidth.

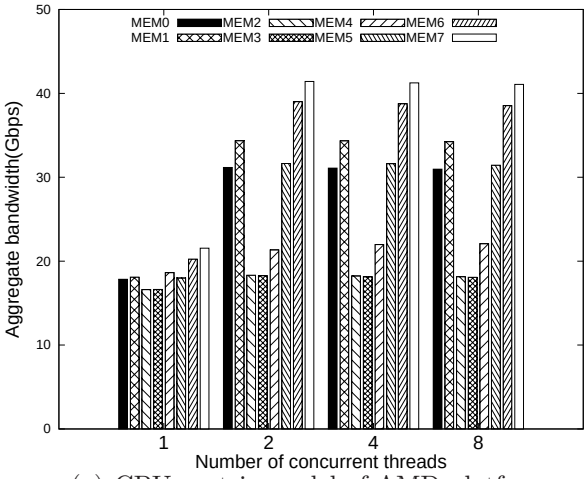
We can then derive the topology of the evaluated AMD host. However, the connectivity inferred from the test data does not match any of the topologies shown in Figure 1-2. We cannot either deduce any reasonable topology due to the performance asymmetry we just mentioned. Therefore, it is inappropriate to simply use physical distance to determine the NUMA cost for modeling memory bandwidth performance.

Observation 2. *The enormous NUMA penalty of remote memory access on the Intel testbed.*

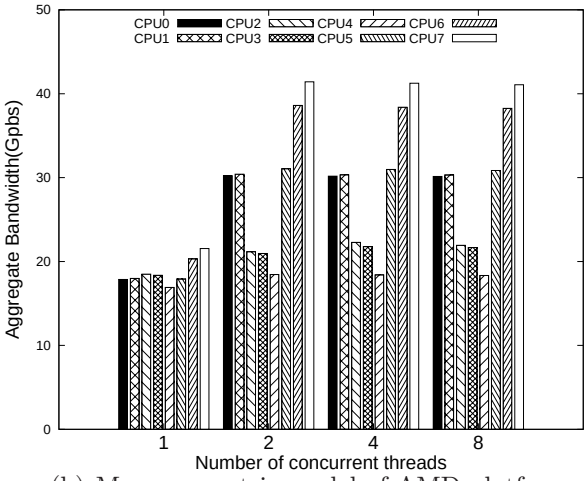
Table 4.2 show the bandwidth NUMA factor of the exemplary node in the Intel platform. Here the bandwidth NUMA factor is referred as the ratio between local memory access bandwidth versus a remote one, which is consistent with the latency NUMA factor we mentioned in Chapter 1. We can see a significant bandwidth performance degradation associated with remote accesses. The NUMA factor reaches as high as 6.63.

4.2.2 I/O performance characterization and analysis

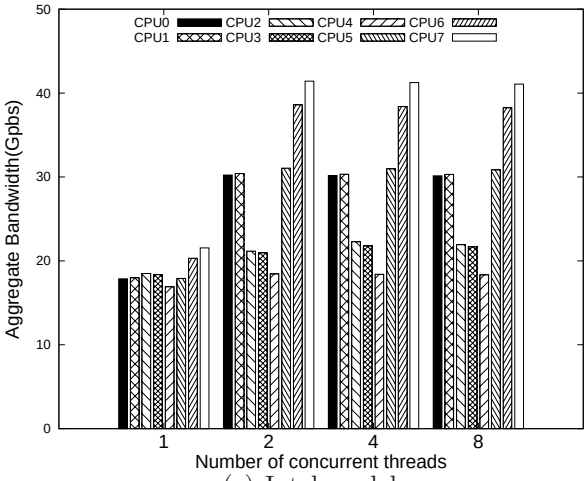
In the previous section, we confirm that hop distance cannot be used as an accurate memory bandwidth performance model, especially for the AMD testbed. Also, an unexpected NUMA factor is also observed on the Intel one. In this section, we will apply the performance model produced by the STREAM benchmark into I/O evaluation, to whether it matches with I/O performance pattern.



(a) CPU-centric model of AMD platform



(b) Memory-centric model of AMD platform



(c) Intel model

Figure 4-3: Bandwidth performance models of AMD and Intel testbeds, produced by STREAM benchmark

As stated in Observation 1, the performance of the AMD testbed is asymmetric and depends on the direction of data flow. Here, we provide a full characterization of the exemplary node of the two testbeds using STREAM benchmark, and show the results in Figure 4-3. Figure 4-3(a) depicts the cases when STREAM benchmark threads run on node 7 to access data on all nodes on AMD testbed. We call this model a "CPU centric" characterization. Figure 4-3(b) illustrates the cases when the data is on node 7 and is accessed by all nodes on AMD testbed. This is called "memory centric". On the other hand, the performance of the Intel testbed, using node 1 as exemplary node, is symmetric and consistent with its system topology. Its CPU centric and memory centric performance patterns are similar to each other. Therefore, we demonstrate both the CPU centric and memory centric models for the AMD testbed, but show only one model for the Intel testbed in Figure 4-3(c).

TCP performance characterization

Here we present the analysis of high-speed network data transfer across the PCIe interconnect to/from a selected NUMA node with both TCP and RDMA protocols. This involves measuring write rate (sending data to I/O devices) and read rate (receiving data from I/O devices). The default NUMA policy in current Linux kernel is *local-preferred*, i.e. the I/O applications can be guaranteed with local memory space if possible. However, this local node may be remote from I/O devices. The application performance model can possibly act as either CPU centric or memory centric model.

Table 4.3 describes the network test configurations. In order to get an accurate long-term bandwidth performance, each data stream is required to transfer 400 Gbytes of data, and the average aggregate performance is then reported. All test cases will allocate buffers in their local memory

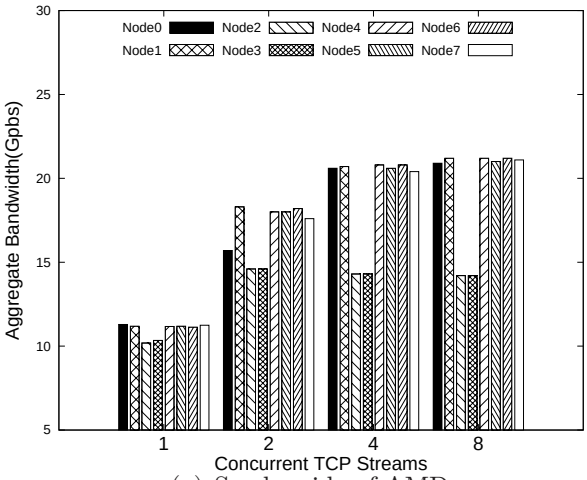
Table 4.3: Parameters for network I/O tests, including TCP and RDMA

Data size requested by each test process	400Gbytes
TCP Variant	Cubic
I/O block size	128Kbytes
I/O depth for RDMA operations	1
Ethernet frame size	9000

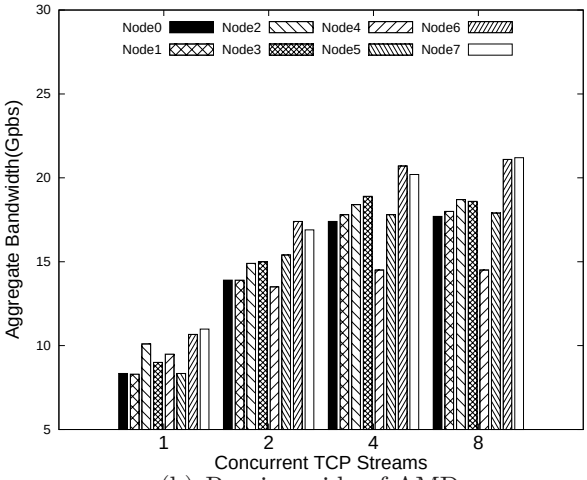
space, and then vary NUMA node where the fio benchmark is executed. Figure 4-4 shows the aggregate bandwidth performance with various numbers of concurrent data streams, and with different NUMA setups on the AMD and Intel testbeds.

For the the AMD testbed, the network adapters use the PCIe Gen 2.0 8x peripheral interface which supports a maximum of 40 Gbps raw transfer rate. Due to the 8/10bit encoding used for the PCIe Gen2 protocol, the available data bandwidth is degraded to 32 Gbps. The real available bandwidth is further decreased by the inherent overhead of network protocols (Ethernet, TCP/IP, RDMA) that support the communication. Therefore, the maximum bandwidth of 25 Gbps in the tests is very close to the theoretical performance limit. On the other hand, the Intel testbed adopts PCIe Gen 3.0 8x interfaces, and can run steadily at 40 Gbps limit while there are more than four concurrent streams.

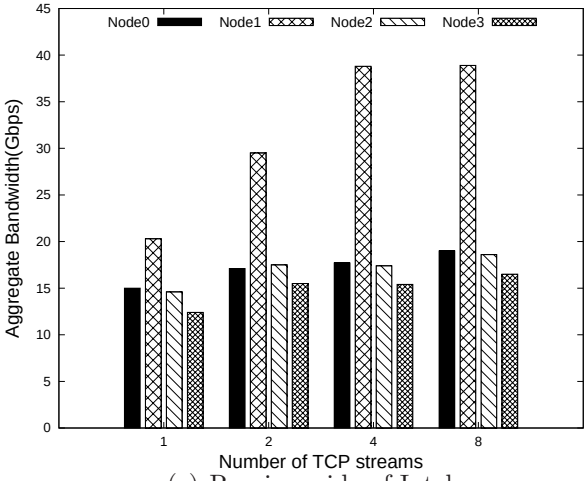
For TCP performance in Figure 4-4, bandwidth grows when the number of concurrent streams increases until there are four parallel streams. When the number of concurrent streams further rises, the contention among them begins to introduce some unexpected behavior. Therefore, sometimes, the performance of node 5 appears to be the best when there are eight or sixteen current threads. Overall, it seems that the TCP send performance in Figure 4-4(a) is close to that in the CPU centric model, while TCP receive performance in Figure 4-4(b) is close to that in a memory centric model.



(a) Sender side of AMD



(b) Receiver side of AMD



(c) Receiver side of Intel

Figure 4-4: TCP bandwidth performance characteristics

Another finding is, when the data path is bound to node 7 locally, the performance is not always the best. In many cases, we can get better performance when we allocate CPU and memory resources on node 6, the neighbor of node 7. This is because all interrupts from I/O device are handled locally by node 7. When we run all application processes on node 7, the contention among multiple tasks will lead to a performance degradation. On the other hand, Node 6 also has its own on-chip access to the I/O devices, and does not have the contention of I/O interrupt handling, and therefore this binding results in an even better performance than the local node 7.

RDMA performance characterization

Observation 3. *Ineffectiveness of STREAM benchmark in both AMD and Intel systems.*

Figure 4-5 shows that the RDMA performance is more stable than that of TCP. That is because RDMA operations offload most of their protocol processings to network adapters, and significantly reduce resource contention. For the test results on the AMD testbed, RDMA_WRITE bandwidth performance is close to that in the CPU centric model, but RDMA_READ does not match with neither the CPU centric model nor memory centric model in Figure 4-3. For example, in both AMD models of Figure 4-3, the performance of node {0, 1} cases is better than that of node {2, 3} cases by 43% to 88%, respectively. When RDMA_READ runs on node {0, 1}, its bandwidth performance is worse than that of node {2, 3} by 15% to 18.4%. By looking at the TCP receiver performance in Figure 4-4 again, we can see that the bandwidth of node {2, 3} cases still slightly outperforms that of node {0, 1} cases. Furthermore, to study the TCP and RDMA performance on the Intel testbed, we calculate the bandwidth NUMA factor while there are eight concurrent streams as Table 4.4.

Table 4.4: Bandwidth NUMA factor of TCP and RDMA operations on the Intel testbed

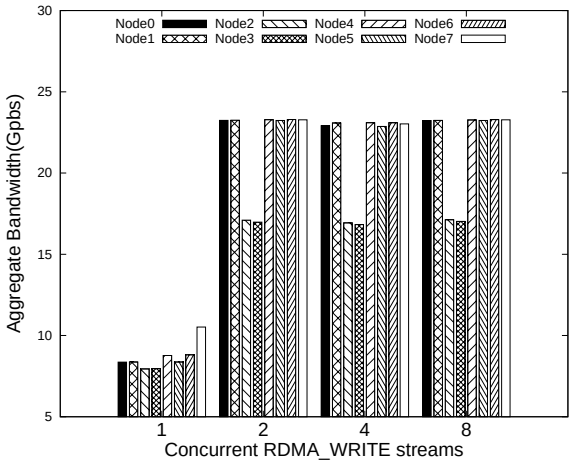
Node ID	0	1	2	3
TCP	2.0	1	2.09	2.36
RDMA_WRITE	1.009	1	1.002	1.314

Compared the data in this table and Table 4.2, the TCP bandwidth NUMA factor is about 3 times smaller than that in the STREAM benchmark case, and for the RDMA_WRITE case it is even 6 times smaller. Similar results can also be summarized from the AMD experiment data. Therefore, we believe that the NUMA factor achieved by the STREAM benchmark overestimates the real one for network I/Os. All of the above indicates that the performance models derived by the STREAM benchmark cannot be applied to network I/O bandwidth patterns.

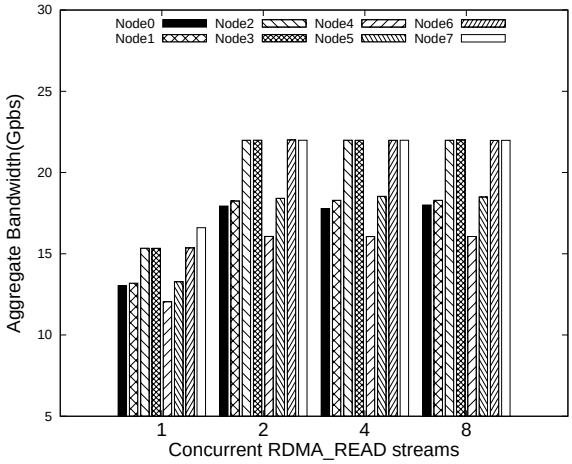
Disk I/O performance characterization

In order to further investigate the performance mismatching of the STREAM benchmark and I/O operations we discovered in RDMA tests, we provide the bandwidth characterization for PCIe-based SSDs using the fio benchmark. We observe that regular kernel-buffered read/write operations perform much worse than kernel-bypassed ones, and asynchronous I/O operations outperform synchronous ones on our testbed. Therefore, we utilize the libaio engine with the kernel-bypass option to maximize transfer speed in the section. In order to further increase the aggregate bandwidth, two LSI SSD cards are accessed simultaneously, and the aggregate bandwidth performance is reported. Hence, the total number of test processes is at least two. Each test process transfers 400 Gbytes of data, with a block size of 128 Kbytes and an I/O depth of 16.

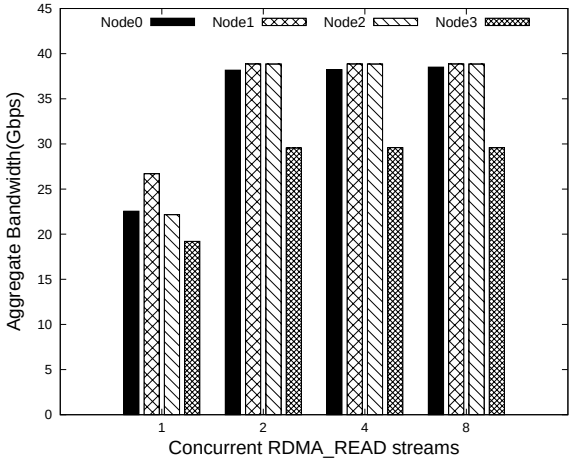
Figure 4-6 depicts the aggregate bandwidth performance with multiple test processes. As expected, the disk write rate closely matches the



(a) RDMA_WRITE on AMD platform

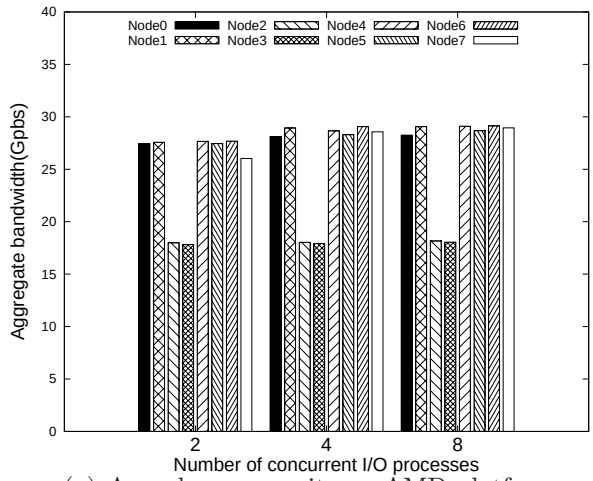


(b) RDMA_READ on AMD platform

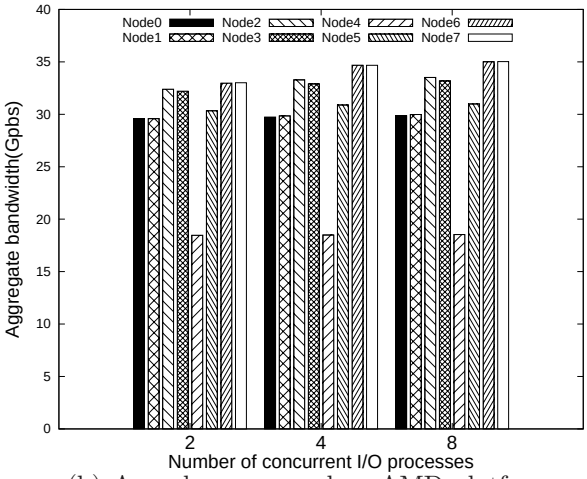


(c) RDMA_READ on Intel platform

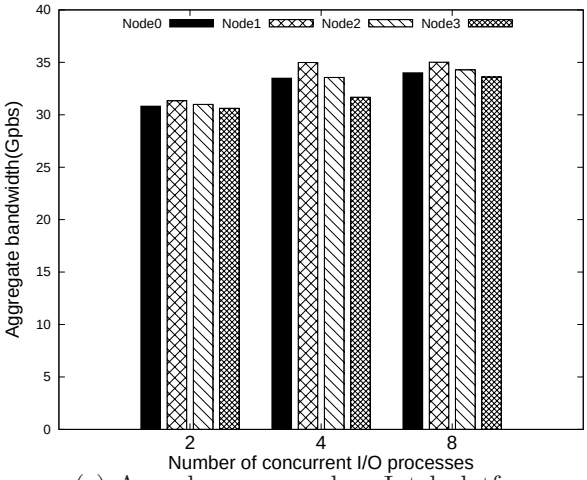
Figure 4-5: RDMA bandwidth performance characteristics



(a) Asynchronous write on AMD platform



(b) Asynchronous read on AMD platform



(c) Asynchronous read on Intel platform

Figure 4-6: Disk I/O bandwidth performance characteristics

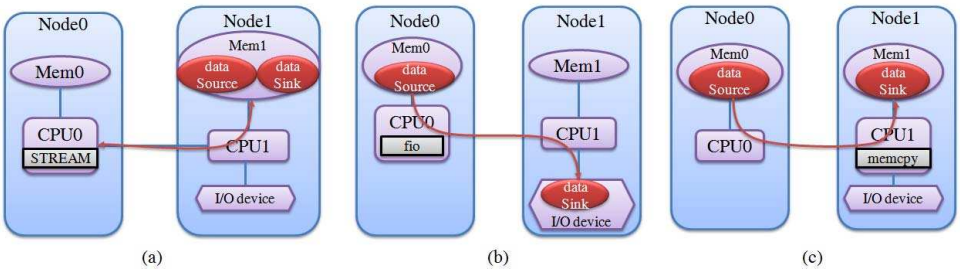


Figure 4-7: (a) Data copy process in STREAM benchmark. Both data source and sink are located in node 1. (b) Data copy process in I/O operations. Data source is in node 0 and sink is in node 1. (c) Data copy process in proposed method. Data source is in node 0 and sink is in node 1.

TCP/RDMA send rate in previous tests, and the disk read rate is close to the TCP/RDMA receive rate. However, none of these I/O performance characteristics is consistent with the STREAM benchmark results in Figure 4-3. The disk I/O bandwidth results in both testbeds confirm that the NUMA factors produced by the STREAM benchmark tests are overstated as well. Meanwhile, we can also observe that the NUMA effects on the AMD 8-node system is more noticeable than that of the Intel 4-node host, and this proves that the NUMA sensitivity grows with the system size, and therefore demands more sophisticated tuning/alignment algorithms to mitigate its negative effects.

4.2.3 Analysis of performance mismatching

All the above results demonstrate that the NUMA characteristics captured by the STREAM benchmark do not match with the actual I/O performance model, especially when application processes read data from I/O devices. This performance inconsistency may be ascribed to the following reasons:

1) **Memory data replication with different message size.** In the STREAM test, the user process requests a large amount of memory to

create and initialize two big arrays of *double* values, and then copies one array to the other, one item at a time. For this small data block transfer, CPU is supposed to utilize its internal Programed I/O (PIO) to move data with its cycles. On the other hand, for bulk data replication between memory and I/O device, CPU will offload the I/O task to a DMA engine, and the actual data replication will therefore bypass CPU. According to the AMD specification [110], the difference of routing methods between accessing CPU core and memory makes us believe that the PIO communication and DMA communication can have distinct paths within involved hardware components.

2) **Data source and sink locations of I/O operations.** The STREAM benchmark itself does not support NUMA-awareness for either the location of program code or the location of allocated memory. To enable it to be aware of the NUMA architecture, we rely on the *numactl* command line tool to statically bind all memory and CPU resources during the entire execution time. Therefore, both data source and sink can only be bound to the same NUMA node, as illustrated in Figure 4-7(a). Meanwhile, for I/O operations, source and sink can be in different NUMA nodes, as shown in Figure 4-7(b). This difference can also lead to different performance characteristics.

4.3 NUMA characterization methodology for I/O operations

In the previous section, we showed that hop-distance and NUMA characterization based on the STREAM benchmark cannot capture I/O performance attributes. In this section, we describe our new method to more appropriately characterizing NUMA effects.

4.3.1 Proposed methodology for the NUMA I/O performance model

We utilize *memcpy* operation and *libnuma* library to move a large bulk of data between a given source memory node and a target memory node. Herein, a target NUMA node refers to the node that is directly attached to I/O devices and needs to be characterized. Without involving the actual data traffic to physical devices, we can simulate the behavior of I/O device's DMA engine with a "memcpy" process that is bound to the target NUMA node and copies large message. Figure 4-7(c) gives an example of this process. The operation of our proposed methodology solves the two mismatched behaviors we mentioned in Section 4.2.3, and therefore can be utilized to build a bandwidth performance model for PCIe based I/O tasks. The detailed procedure of our proposed methodology is shown in Algorithm 1. It first finds out the number of parallel threads. In order to take full advantage of the CPU cores in each NUMA node, the number of parallel threads is set to be the number of CPU cores in the node. Then multiple test threads are initiated to copy data simultaneously and independently.

For PCIe based I/O operations, CPU offloads data replication tasks to the DMA engine in I/O devices. For data write cases, the DMA engine reads data from the host memory, and stores it into the buffers inside I/O devices. In data read cases, DMA reads data from the buffers in the I/O hardware, and writes to the host memory. To simulate these scenarios, we enforce all the data-copy threads running on the target node to simulate a DMA engine in I/O devices. In the simulating cases of writing into I/O devices, the data sink is statically bound to the target node and the source node varies among the tests, as shown in Figure 4-8(a). In the reading test cases, we fix the source to the target node and vary the data sink node, as

Algorithm 1: NUMA I/O performance modeling

```
   Input: NODES TO CHARACTERIZE: k, MODEL TO OBTAIN: model
 1 n ← numa_num_configured_nodes()
 2 m ←num_configured_cores() / n
 3 for i ← 1 to n do
 4     for p ← 1 to m do
 5         if mode = write then
 6             Allocate memsrc[p] in NUMA node i
 7             Allocate memsnk[p] in NUMA node k
 8         if mode = read then
 9             Allocate memsrc[p] in NUMA node k
10             Allocate memsnk[p] in NUMA node i
11     for p ← 1 to m do
12         Create thread[p], bind to node k, copy from memsrc[p] to
           memsnk[p] for 100 times and record the average bandwidth
13     for p ← 1 to m do
14         thread_join(thread[p]);
15 if mode = write then
16     Generate I/O device write performance model for node k
17 if mode = read then
18     Generate I/O device read performance model for node k
19 return
```

illustrated in Figure 4-8(b). In both reading and writing cases, the data source node and the data sink node can be the same. In this way, without involving I/O devices, we can emulate the I/O data replication with only memory copy operations, and learn the I/O performance characteristics without costly I/O benchmark tests.

Figure 4-9 shows the CPU centric (I/O device write) and memory centric (I/O device read) bandwidth performance with the proposed methodology on both platforms. For the performance models of the AMD platform, we can see that this result matches the I/O bandwidth performance levels we showed in Section 4.2.2. Some performance differences captured by our tool are not reflected in I/O test results. The reason is that, as stated in

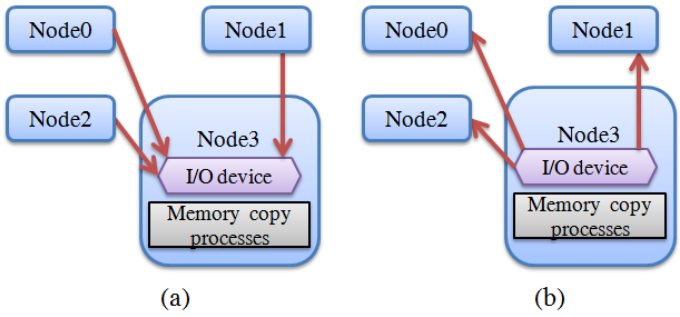


Figure 4-8: Simulate I/O behavior with memory copy operation. Assume that there are 4 nodes in the system and I/O device is attached to node 1, so the target node for testing is node 1. (a) I/O device write simulation. Data sink locates at node 1, and data source varies among test cases. (b) I/O device read simulation. Data source locates at node 1, and data sink varies among test cases.

Section 1.2, the I/O bandwidth performance can be impacted by various factors. In these cases, the I/O bandwidth bottleneck is not related to the NUMA penalties. We categorize all the nodes into different classes in Table 4.5 as the device write model and Table 4.6 as the device read model, according to their relative performance levels in Figure 4-9. The local and neighboring nodes are always be assigned to the first class, and the main task of our methodology is to classify remote nodes. The performance models in both tables were first obtained by the proposed method, and then used to compare and analyze the actual I/O performance characteristics.

We also compute the bandwidth NUMA factor of the Intel platform as Table 4.7. Compared with the STREAM benchmark results in Table 4.2, the NUMA asymmetry measured by our proposed method significantly decreases, and it is more close to the real I/O performance characteristics in Table 4.4.

In conclusion, the proposed methodology successfully solved the problems described in the three observations, and is more effective in characterizing the I/O performance compared to other popular NUMA-related

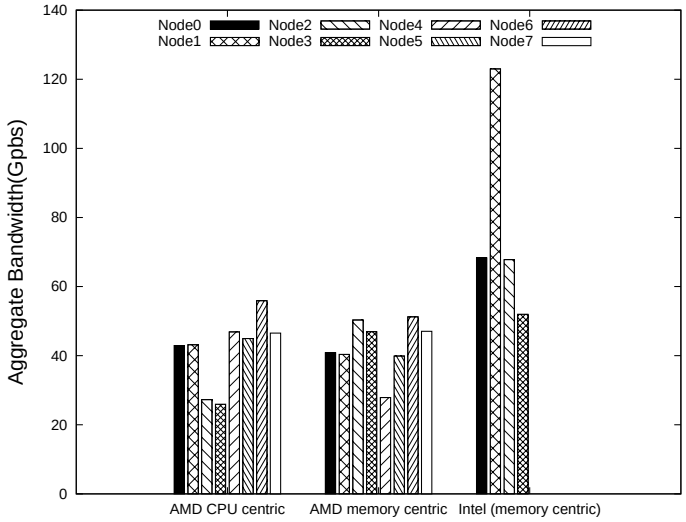


Figure 4-9: Bandwidth performance model of the exemplary nodes on both evaluated systems, produced by the proposed methodology

Table 4.5: NUMA I/O bandwidth performance model for device write(Unit:Gbps)

Operation	Node ID	Class 1 6, 7	Class 2 0, 1, 4, 5	Class 3 2, 3
Proposed memcpy	Range Avg	46.5 – 55.9 51.2	42.9 – 46.9 44.5	26.0 – 27.3 26.6
TCP sender	Range Avg	19.6 – 20.9 20.3	20.0 – 21.0 20.4	16.2 – 16.3 16.2
RDMA_WRITE	Range Avg	23.3 – 23.3 23.3	23.2 – 23. 3 23.2	17.0 – 17.1 17.1
SSD write	Range Avg	28.6 – 29.1 28.8	28.1 – 28.9 28.5	17.9 – 18.0 18.0

metrics and benchmark methods.

4.3.2 Implementation and application of the proposed method

The methodology used to model the performance of exemplary nodes on the two platforms can also be generalized to other nodes in the hosts and also to other NUMA systems. The model method was implemented as

Table 4.6: NUMA I/O bandwidth performance model for device read(Unit:Gbps)

Operation	Node ID	Class 1 6, 7	Class 2 2, 3	Class 3 0, 1, 5	Class 4 4
Proposed memcpy	Range Avg	47.1–51.2 49.1	46.9–50.3 48.6	39.9–40.9 40.4	27.9 27.9
TCP receiver	Range Avg	20.3–22.0 21.2	19.6–20.4 20.0	19.8–21.1 20.6	14.4 14.4
RDMA_READ	Range Avg	22.0–22.0 22.0	22.0–22.0 22.0	18–18.5 18.3	16.1 16.1
SSD read	Range Avg	34.7–34.7 34.7	32.3–32.9 33.1	29.7–30.9 30.1	18.5 18.5

Table 4.7: Bandwidth NUMA factor of the Intel platform using the proposed method

Node ID	0	1	2	3
NUMA factor	1.8	1	1.81	2.37

an *iomodel* test module, and added into the standard *numademo* software package. The proposed module can automatically characterize the nodes that are directly attached to I/O hubs, or specified by users. In addition to providing a comprehensive representation for a full CPU/memory centric performance matrix, as shown in Figure 4-9, the module can also generate a performance model for I/O device access as a form, illustrated as the following Listing 4.1.

Listing 4.1: Text output of the proposed module on Node 7 of the AMD testbed

```
8 nodes available
32 CPUs available
Number of element in tested matrix = 21845333
Required memory size=524288000
Number of Threads used in this profiling = 4
Node 7 IO model:
```

```
CPU Centric levels:
        Class1: 7 6
        Class2: 0 1 4 5
        Class3: 2 3
Memory Centric levels:
        Class1: 7 6
        Class2: 0 1 5
        Class3: 2 3
        Class4: 4
```

The obtained performance models from our methodology and software can then bring in the following advantages:

1) **Reduce the cost to capture the NUMA characteristics of the entire system.** In the performance model, we assume that all the nodes in the same class have similar bandwidth performance. While we want to understand the NUMA impact on system performance, instead of benchmarking all possible combinations, we can examine only one node from each class. For example, in the case of Table 4.6, if we characterize the read speed of I/O devices attached to node 7, we only need to test four different NUMA configurations, one from each of the four classes. These representative tests will provide the same results as the entire set of test cases (eight cases). Hence, the evaluation cost decreases by 50%. This brings more benefits when the system becomes larger and integrates more NUMA nodes.

2) **Predict the overall performance in multi-user environment.** When an I/O device is shared by multiple users, it is highly possible that data-access requests come from different NUMA nodes. Assume that we have achieved the performance model for the node attached to the I/O device using our method. Let BW_i be the average bandwidth performance of Class i in the performance model, and $\alpha_i\%$ be the percentage of data accesses that come from Class i, where $i \in [1, \cdots, N]$, and N

is the total number of classes. We predict the aggregate performance for I/O devices using the following model 4.1, $\hat{BW_{io}}$. For example, in the case of RDMA_READ in Figure 4-5(c), if two processes transfer data from node 2 (of class 2) to the network card in the system, and two other processes access from node 0 (of class 3), the overall bandwidth is estimated as $\hat{BW_{io}} = 50\% \times 18.036 + 50\% \times 21.998 = 20.017 Gbps$. We run this configuration with the fio benchmark. Since the bandwidth performance is stable over the whole data replication process, instead of showing the distribution of multiple repetitive test results, we demonstrate the average aggregate bandwidth while transferring a large amount of data (400 Gbytes per process), which is 19.415 Gbps. Hence, the relative error is $\varepsilon = \frac{|20.017-19.415|}{19.415} \times 100\% = 3.1\%$. This example shows that our model can estimate the overall performance of I/O devices in a multi-user scenario.

$$\hat{BW_{io}} = \sum_{i=1}^{N} \alpha_i\% \times BW_i \tag{4.1}$$

3) **Assist resource schedulers on NUMA systems.** To design an application-layer NUMA scheduler, a programmer should have enough information about a system's NUMA characteristics. With the help of our benchmark tool and modeling method, numerous scheduling algorithms [111–113] can be applied to modern high-end NUMA systems. For example, in a multi-user environment, binding all I/O tasks to their local node will lead to severe performance degradation due to the contention of shared resource. With the knowledge of our performance model, the task scheduler can distribute application processes to nodes in the same class or the classes with the same performance. For example, in the case of RDMA_WRITE in Figure 4-5(a), based on our characterization, we know that class 1 and class 2 have almost identical performance. Therefore, instead of allocating all application processes to node 7 only, we can evenly split the task processes

among all nodes in Class 1 and 2. Therefore, the contention over local resource is greatly relived, and the overall performance can be improved.

4.4 Summary

In this paper, we addressed the problem of accurately characterizing and predicting I/O bandwidth performance in modern high-end NUMA systems. Directly applying a software benchmark to characterize memory and I/O hardware might lead to unexpected performance inconsistency among multiple tests, and can potentially generate a large amount of workload. We illustrated the reasons why existing NUMA-related metrics and tools cannot address the problem by quantitative comparisons. We then proposed our own methodology and software to obtain an accurate I/O bandwidth performance model without involving the physical I/O hardware and time-consuming I/O benchmarking. To the best of our knowledge, this work is the first attempt to propose an I/O performance model based on simple memory operations for NUMA systems. The experimental results confirmed that our empirical method can effectively predict the I/O bandwidth characteristics among various NUMA architectures. At last, we demonstrated the significance of our methodology by providing three concrete examples that can take advantage of the performance models generated by the proposed tool. This performance model can be imported to data replication applications to support efficient NUMA scheduling. This scheduling mechanism is a good alternative to the dynamic scheduling method we proposed in last Chapter 3.

Chapter 5

Multicore Resoure Scheduling for Data Replication

Most existing NUMA-aware design in data intensive applications is to maximize data and I/O device locality. However, maximizing locality does not always deliver maximum performance, as illustrate in Section 1.1.1. An effective scheduling design should also avoid the penalty of contention. While there is intensive contention in local NUMA node, the scheduler needs to dispatch threads/data to other remote nodes with minimal access cost. This is often a challenging task in both mathematical and empirical studies. This chapter first explores the mathematical solution of finding the optimal thread allocation and task mapping on NUMA multicore system, and analyzes its complexity. For empirical solution, we implement a thread scheduling module in BBCP software, and evaluate its benefits under different levels of contentions, using the high performance testbed described in Section 3.1. This illustrates the advantages of adding NUMA-awareness to data replication applications.

Table 5.1: List of notations used in problem formulation

$D(u)$	The total size of requested data on storage node u
C	The maximum number of I/O threads running on each NUMA node without degrading the overall performance
$\beta_d(u,v)$	The total size of data scheduled to be transferred from node u to node v by disk I/O
$\beta_n(v,w)$	The total size of data scheduled to be transferred from node v to node u by network I/O
$\gamma_d(u,v)$	The number of disk threads scheduled to be created to copy data from node u to node v
$\gamma_n(v,w)$	The number of network threads scheduled to be created to copy data from node v to node w
$B_d^0(u,v)$	The bandwidth of a single disk I/O threads between node u and node v
$B_n^0(v,w)$	The bandwidth of a single network I/O threads between node v and node w
$B_d^{max}(u,v)$	The bandwidth limit for disk I/O between node u and node v
$B_n^{max}(v,w)$	The bandwidth limit for network I/O between node v and node w
$t_d(u,v)$	The total execution time of sending all the requested data from node u to node v
$t_n(v,w)$	The total execution time of sending all the requested data from node v to node w

5.1 Mathematical Model

5.1.1 Problem formulation

The detailed notion of this section is listed in Table 5.1. Consider an end system with N NUMA nodes, and each NUMA node connects to its own local storage system and network devices. Each storage can have single/-multiple tasks that need to be transferred out via network adapters. Without loss of generality, we divide this data sending process into two steps, disk I/O and network I/O, as shown in Figure 5-1. The requested data in the storage attached to NUMA node u, are first passed to system memory node v by disk I/O threads, and then send out via network interface on

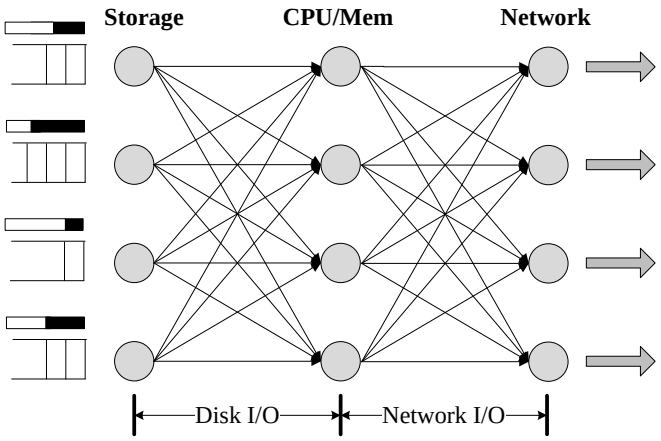


Figure 5-1: Graph model of NUMA scheduling problem at data sender

node w by network threads, where $u, v, w = 1, 2, \cdots, N$.

In addition, we assume that the aggregate bandwidth increases linearly with the number of threads between two nodes, until reach the bandwidth limit of the link. Therefore, our objective is to create optimal number of I/O threads, and assign optimal data size to each thread, which minimizes the execution time of finishing all the data replication tasks. In this two-stage pipelined process, the execution time of each data replication task is determined by the longer operation time between disk I/O stage and network I/O stage. Therefore, this optimization problem can be modeled as a min-sum-max resource allocation problem (MSMRAP) shown in Model 5.1-5.6.

$$min_{\gamma,\beta} \sum_{u=1}^{N}\sum_{v=1}^{N}\sum_{w=1}^{N} max(\frac{\beta_d(u,v)}{\gamma_d(u,v)\cdot B_d^0(u,v)}, \frac{\beta_n(v,w)}{\gamma_n(v,w)\cdot B_n^0(v,w)}) \tag{5.1}$$

$$\text{Subject to} \qquad \sum_{v=1}^{N} \beta_d(u,v) = D(u), \qquad u = 1, 2, \cdots N \tag{5.2}$$

$$\sum_{u=1}^{N} \beta_d(u,v) = \sum_{w=1}^{N} \beta_n(v,w), \qquad v = 1, 2, \cdots N \tag{5.3}$$

$$\sum_{u=1}^{N} \gamma_d(u,v) + \sum_{w=1}^{N} \gamma_d(v,w) \leq C, \qquad v = 1, 2, \cdots N \tag{5.4}$$

$$\text{Over} \qquad 0 \leq \gamma_d(u,v) \leq \left\lceil \frac{B_d^{max}}{B_d^0} \right\rceil, \quad 0 \leq \gamma_n(v,w) \leq \left\lceil \frac{B_n^{max}}{B_n^0} \right\rceil \tag{5.5}$$

$$\beta_d(u,v) \geq 0, \quad \beta_n(v,w) \geq 0 \tag{5.6}$$

5.1.2 Computational complexity analysis

Although all the constraints are linear in the derived MSMRAP problem we described in section 5.1, solving this problem still incurs significant computational complexity. The reasons are as follows.

- **Mixed integer problem.** There two sets of variables in the problem. One is the data size passed between nodes, $\beta_d(u,v)$ and $\beta_n(v,w)$. They are continuous variables. The other set is number of worker threads, $\gamma_d(u,v)$ and $\gamma_n(v,w)$. They are constrained to be integers. Therefore, the target problem is classified as a Mixed Integer Programming (MIP) problem. It becomes NP-complete if the value space

of the integer variables is infinite.

- Complexity of MSMRAP. By applying appropriate relaxation methods, such as branch-and-bound and cutting plane, we may be able to transform all the variables to be continuous. However, even with continuous variables, the derived MSMRAP problem is still computationally intactable. As proven in the study [114], the problem can be transformed to CLIQUE problem, which is still NP-complete. Moreover, the item in the objective function is fractional, which introduce another dimension of complexity.

5.1.3 Divide and conquer solution

The previous Section 5.1.2 shows that the resulted MIP MSMRAP problem is NP-complete. Nevertheless, there are only two terms in the objective function, we can add an additional constraint to divide the problem into two Sum-of-Ratio subproblems, i.e. disk-determined one and network-determined one. However, the Sum-of-Ratio problem is still NP-complete, as proven in literature [115]. We need further simplification to the problem.

In practice, the value set of the discrete variables, $\gamma_d(u, v)$ and $\gamma_n(v, w)$ is usually small. Firstly, the upper limit of these variables depends on the total number of cores in a CPU package, which is usually less than 12 in the state-of-the-art commodity servers. More importantly, for current high speed disk and network devices, eight parallel threads are usually enough to reach their optimal performance with proper system tuning. Finally, the value set of discrete variables can be further reduced by adding more practical constrains. For example, if there is no requested data on one storage node, we do not need to assign any disk I/O threads accessing this node. Another example is that, while there are some disk threads copying data to one memory node, there much have at least one network thread to

carry the data out. This is called thread conservation rule.

Therefore, after shrinking the feasible solution set, we can enumerate the possible data path exhaustively with proper path searching algorithms. After this, the original problem can be divide into multiple Linear Programming(LP) subproblems. The global optimal solution can be obtain after solving all these LP problems. Compared to the approximation algorithms and heuristic algorithms, the proposed divide-and-conquer algorithm has following advantages. First, the global optimal solution is guaranteed. Second, there are many existing algorithms and tools can solve these LP problems very efficiently. Third, the resulted independent LP problems are be processed parallelly on multicore system, which further reduces the overall execution time. However, the computation complexity of the proposed algorithm is not guaranteed to be polynomial, and can become NP-hard while the value set grows.

5.2 NUMA-aware BBCP Implementation and Evaluation

In last Section 5.1, we show the significant mathematical difficulty of solving resource scheduling problem on NUMA multicore systems. This section turns to implement practical thread/memory scheduling module in real world data replication software, and evaluate its effectiveness on the state-of-art testbeds.

Enabling multicore-awareness, especially NUMA-awareness, is also non-trivial for real world data replication software. Most of existing high-speed data replication tools do not take multicore technology into consideration. Even though many of them use multiple threads for high performance parallel data replication, these threads are not aware of the underlying NUMA

architectures, and subject to the default OS kernel scheduling. The performance of data replication tasks highly depends on the run-time scheduling, and this may result in the aforementioned performance gap in Section 1.1.1.

In this section, the BBCP data movement tool is chosen as the representative of high performance data replication software: First, it serves as a multi-threaded extension of *scp*, the ubiquitous data replication tool in all Linux distributions. Anyone can easily use and understand its interface, and take advantage of the new high performance feature. Second, BBCP is widely used in data-intensive science programs and petascale supercomputing centers. Multiple recent studies [116–118] have proved its capability to transfer data at line speed. Third, runtime NUMA-awareness is especially important for point-to-point data replication applications, of which BBCP is a good reference example. The detailed reason of this will be explained in Section 5.2.1.

5.2.1 Implementation of resource scheduling module

Figure 5-2 shows that three different entities are involved during the BBCP data replication process: a source node, a target node, and a control agent. The control agent is in charge of initiating a data replication by instantiating a source node and a target node, and then requesting the target to fetch data from the source. The control agent is also responsible for relaying all necessary parameters to the source and target nodes for coordinating data replications. These three entities are launched from a single command on the server that hosts the control agent, and usually distributed on different servers. Hence, it is impossible to simply use the *numactl* command-line tool to pin all the entities to the pre-selected nodes on all the servers. To solve this problem, we implement a NUMA-aware module and integrated it into BBCP to enable automatic NUMA configurations at runtime and thereby ensure the resource affinity.

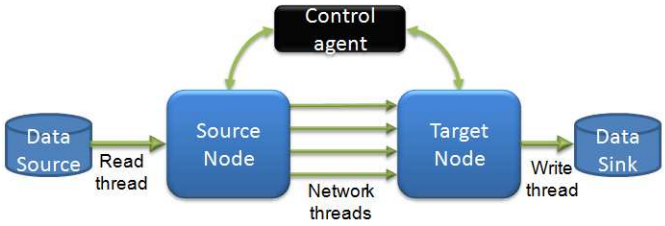
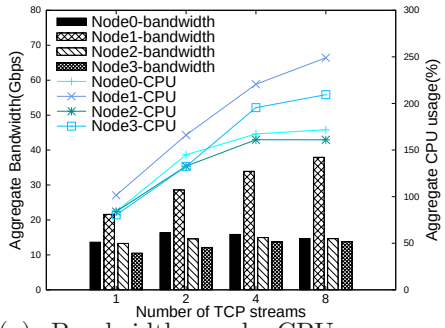


Figure 5-2: Communication between BBCP entities

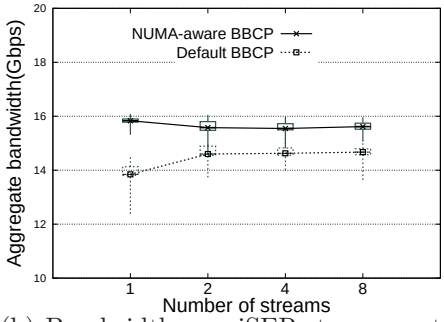
BBCP software supports user-defined configurations for I/O block size, TCP window size, the number of TCP data streams, etc. As shown in Figure 5-2, four parallel TCP streams are employed as the default setting, and four network threads are initiated at both the source and target nodes, each of which manage one TCP session. All of them share one thread for disk read/write. Our integration utilizes the *libnuma* library [94] to control the placement of these running threads and their corresponding memory, respectively. During the initiation stage, both source and target nodes will detect the affinity information of network adapters to be used for transfer. When the threads for TCP streams are created, they will be enforced to run on the nodes that are directly connected to the involved network adapters. With this configuration, we ensure the affinity among all threads belonging to a data replication task and network adapters. Therefore, the application can achieve the best NUMA locality automatically. The NUMA scheduling module also provides an interface for user to define NUMA configurations and the placement of the source and target nodes.

5.2.2 Evaluation on high performance testbed

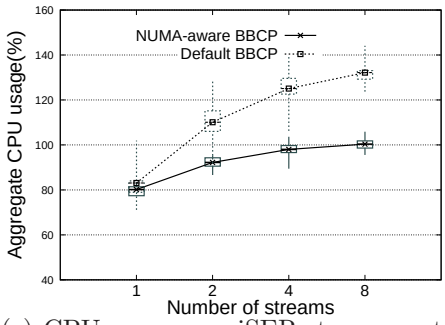
In this section, the performance of NUMA-aware BBCP application is characterized on our evaluation testbed. TCP window size and I/O block size are all set to 512 Kbytes, an optimal configuration for BBCP on the testbed which is learned from concrete experiments. Moreover, locally attached



(a) Bandwidth and CPU usage of memory-based tests



(b) Bandwidth over iSER storage system



(c) CPU usage over iSER storage system

Figure 5-3: BBCP performance over 40 Gbps network link

disks are the performance bottleneck, and any test involving local disks will not be able to show the effect of NUMA optimization to network I/O tasks. To eliminate this bottleneck, we use the memory-based tests where data will be read from "/dev/zero" at the sender server, and copied into "/dev/null" at the receiver server. Alternatively, SAN-based device that aggregates disk bandwidth is also used as the data sink in the evaluation system, as shown in Figure 3-2.

Figure 5-3(a) shows the measured results of BBCP over a single 40 Gbps Ethernet link. All tests here were done during a time period of 300 seconds. All threads are bound to various CPU nodes with the proposed thread scheduling module. The performance correlates well with the iperf tests in the previous section. The local binding consumes more CPU resources than the remote ones because they can transfer up to 200% more data across network in the same time. This confirms the effectiveness of the proposed

module for users to specify any affinity settings with the modified version of BBCP.

The result of SAN-based tests is shown in Figure 5-3. This box plot shows the inter-quartile range of the distribution of a set of data points. The box contains values between the 25^{th} and the 75^{th} percentiles for that set of data points. The line in the box denotes the mean value. The two ends of the "whiskers" are the upper and lower bound respectively. Here, each test is set to run 35 times, and thus each data set has 35 values. As shown in this figure, via automatically guarantee the best affinity along the data path, NUMA-aware BBCP consistently achieves higher bandwidth with lower CPU usage than the original version of BBCP in all test cases. Furthermore, the smaller areas covered by the NUMA-aware BBCP boxes represent that its performance is more stable and consistent, compared to the original BBCP. The bandwidth performance cannot reach the maximum capability of the network hardware, 40 Gbps, due to the bottleneck of single-threaded disk I/O implementation in the BBCP software. However, as a proof of concept, we can still observe the advantages of application level NUMA-awareness over the OS default one.

5.2.3 Exploring the behavior under contention

In a multi-user and multi-task environment, the contention for various shared resources is inevitable, and has profound impacts on the performance of data intensive applications [17]. In this section, we evaluate the performance difference between the NUMA-aware and original BBCP in the presence of resource contention. As mentioned in previous sections, without application-level NUMA optimization, BBCP relies on the OS default thread-independent scheduler for assigning CPU core and allocating memory. For the NUMA-aware BBCP, all threads are forced to run on the nodes that are local to network adapters.

Two parallel Ethernet links, with an aggregate bandwidth of 80 Gbps, are used in this experiment, as shown in Figure 3-2. To take the full advantage of two physical connections, two BBCP instances are launched in each test run, with one data replication over each link. Moreover, multiple STREAM benchmark threads are used to create contention for the shared resources inside the NUMA node. We use the STREAM benchmark here because its behavior is easily understood, and it facilitates the performance analysis. During this experiments, various number of STREAM threads would be statically pinned to each NUMA node associated with network adapter, i.e., node 0 and 1 in our case.

Figure 5-4 shows the aggregate bandwidth performance without any contention, and with the contention of four or eight concurrent STREAM threads. With multiple tasks competing for intra-node resources, both NUMA-aware and NUMA-unaware cases experience performance degradation. However, the default OS scheduler only considers load balancing instead of hardware asymmetry, and could result in dispatching a thread to an inappropriate remote core, and possibly migrating threads from the local core to an inappropriate remote one, and thereby causing random results and performance degradation. With the NUMA-aware thread binding strategy, the BBCP bandwidth performance becomes more stable, and improves by 5.93% to 10.83% without any contention, and by 10.5% to 219.56% under resource contention, compared with the standard BBCP. We notice that the performance improvement with no contention is not significant, especially when there are a small number of threads involved. This is due to the fact that there are a sufficient number of cores catering the existing threads in these cases, and the default OS scheduler tends to dispatch threads to the CPU node which is attached to network adapters. However, when there are contention workloads in CPU nodes, the load balancing service will assign a newly spawned thread, or migrate a run-

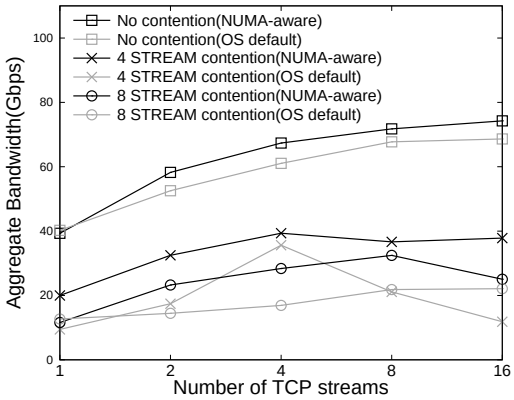


Figure 5-4: BBCP bandwidth performance under multiple levels of contention (higher is better)

ning thread, to an improper remote NUMA node, regardless of NUMA access pattern in the system. This leads to performance degradation and instability.

Finally, another notable observation is that the maximum performance difference appears when there are four competing STREAM threads. When the number of contention thread increases to eight, the benefits of local thread mapping diminish with massive data replication threads because of the intra-node contention for shared resources, such as CPU cores, on-die memory controller and system request queue. This indicates that a good thread scheduler should be both NUMA-aware and contention-aware.

5.3 Summary

In the chapter, we formulate the NUMA thread mapping problem into a MSMRAP mathematical model. A complexity analysis is provided and possible solution is given. However, the problem is still NP-complete after adding multiple relaxations. We then turn to empirical methods for solving the resource schedule problem on NUMA multicore platforms for data replication applications. We implement a resource mapping module for the

BBCP data replication application to enhance it with the NUMA-aware optimization. Experiments with our high-performance testbed demonstrate that the NUMA-aware BBCP achieves significant throughput improvements over the original BBCP implementation, i.e. 10.83% to 220% in the memory-based tests and 6.45% to 14.3% in the SAN-based tests. This confirms the effectiveness of the proposed NUMA-aware optimization, and provides a good hint for the design of high performance data replication applications.

Chapter 6

Resource-Aware Asynchronous Data Replication with Multicore Systems

Previous chapters of the dissertation detailed the importance, quantification and modeling of NUMA effects and NUMA-awareness for I/O. Chapter 3 also gives an example of NUMA-aware data transfer implementation. However, this implementation does not deliver much advantage while there is not competing processes. Except for NUMA-awareness, achieving consistent high performance over various data transfer workload requires deeper software redesign and intensive optimization integrations. The work in this chapter proposes an entire new data transfer solution, including framework and detailed implementation, using multi-threading, asynchrony, event-driven and resource-aware design. It then provides comprehensive analysis and evaluation of various optimizations integrated, including NUMA-awareness. Finally, we compare the performance of the proposed solution with multiple popular software systems in high performance computing community and industry with state-of-art real nation-wide testbeds. The

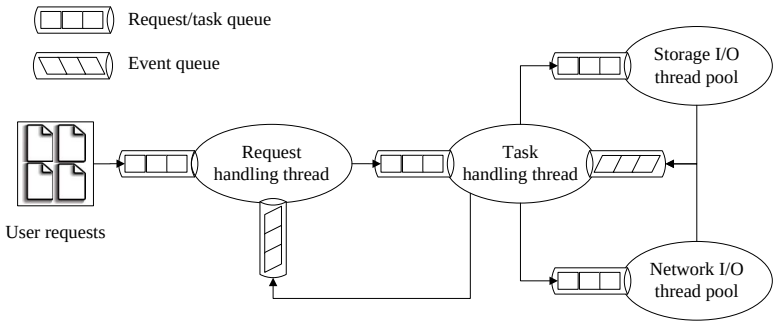


Figure 6-1: Design of asynchronous task processing and data flow

results confirm the significant advantages of the new design for various practical workloads.

6.1 Framework and Protocol Design

An inherent weakness of most traditional data transfer software design is that they use single-threaded sequential processing, as demonstrated in Section 1.1.2. To scale up the end-to-end performance for large datasets, data transfer applications often fork multiple identical sessions, and involve data-intensive operations that overburden CPU cores, memory, and I/O subsystems. To disperse load stress and match the distributed nature of modern multicore systems, we must substantially revise the sequential design, and introduce asynchronous and multithreaded mechanisms to parallelize the entire data transfer processing pipeline.

To address this problem, in Figure 6-1, we present a simplified version of the data flow. It divides an end-to-end data path into a series of stages, and spawns dedicated threads for each of them. Explicit task/event queues are allocated to connect these stages, and thread synchronization primitives are used to ensure concurrent accesses. Thereby, the most salient feature of this design is that threads from different stages execute asynchronously.

Figure 6-2 depicts the four-layered framework design of the proposed

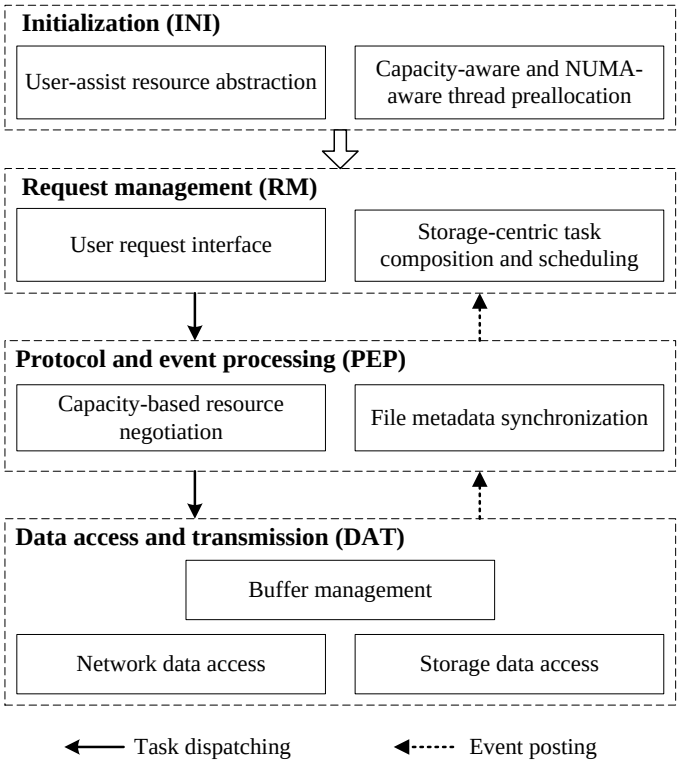


Figure 6-2: A schematic overview of the proposed framework

multicore-aware data transfer solution. When the system starts, the initialization layer executes once to preallocate the resources. The remaining three layers comprise the pipeline for handling data transfer requests. This layered architecture allows us to present the function components systematically. This section first discusses the supported features, and follows with the description of each layer in the proposed framework.

6.1.1 Features for Ensuring High Performance Transfer

Asynchronous processing. The proposed data flow model is I/O resource-centric. All data transfer tasks and system I/O devices have their dedicated thread pools and data structures for managing the involved entities. This approach naturally separates task management, network and storage I/O

processing. Each stage in the model selects a combination of pipelined, concurrent, and event-driven operations to maximize resource utilization. The whole data flow design can be implemented as a daemon process. All the threads and memory are allocated and scheduled via a centralized control, and the threads execute in an asynchronous fashion.

Storage-centric design. The proposed design improves the performance of storage systems, usually the bottleneck along the end-to-end data transfer path. We treat each storage unit adaptively via storage-aware thread preallocation and storage-centric task scheduling. Data transfer tasks are grouped by their targeted storages. Here each storage relies on a pool of pre-allocated I/O threads that are statically bound to the affinitive cores to serve relevant tasks. This method facilitates orchestrated accesses to storages, and also enables other potential optimizations, for example, file-level sorting described later.

Capacity-based scheduling. The bandwidth capacity of I/O devices is another key aspect to be considered in our design. The resource allocation module first assigns threads according to the device's bandwidth capacity. Subsequently, the capacity of a data transfer task is decided and negotiated by the proposed protocols in Section 6.1.4. To determine a task's capacity actually constitutes pinpointing the I/O bottleneck of the entire data transfer pipeline that involves both the data source and sink. Lastly, the scheduling module allocates the thread and memory resources to a given task based on the capacities of the task and I/O devices involved. Conclusively, this capacity-based design takes into account the requirements of individual tasks when assigning pre-allocated resources.

6.1.2 Initialization (INI) Layer

The initialization layer profiles all system I/O resources, and creates an abstraction and a description of major characteristics for each one. For ex-

ample, for each network interface, it contains the network's logical name, address, and physical bandwidth capacity; for a storage device, it includes the storage type (HDD, SSD, or memory), partition name, volume, and also the bandwidth capacity. We design a user-assisted abstraction wherein users can guide the abstraction process via a static configuration file. For example, a network interface can be used for a network connection for sending and receiving traffic, or for an external storage adapter such as Lustre [119]. Accordingly, the users or system administrators need to inform the transfer application of the exact purpose of an interface when an automatic detection mechanism cannot easily resolve it. This configuration file can also takes the performance model generated by the proposed method in previous Chapter 4, and the model can then be used to support NUMA-aware resource mapping and migration.

The second purpose of the initialization layer is to create the management components for each resource. All network I/O threads (senders and receivers) and storage I/O threads (readers and writers) are pre-allocated, and then grouped into the thread pools, each of which is attached to the corresponding abstraction object of an I/O device. The number of I/O threads per pool is determined by the type of the associated device and its bandwidth capacity. The higher its bandwidth is, the more threads the device's pool obtains. For example, multiple I/O threads will be allocated to the storage device that has a good random access performance, e.g., SSD drives and memory-based storages (NVRAMS). In contrast, only a single thread is assigned to the device that performs better on sequential accesses, e.g. HDD drives. Meanwhile, all threads belonging to the same I/O device are created and bounded to assure affinity to the same NUMA node to which the I/O device is attached. Therefore, this capacity-aware and NUMA-aware preallocation module guarantees pertinent resource preparation while enforcing localized thread binding. Furthermore,

by pre-allocating and recycling I/O threads upon completion, we avoid the overhead of dynamically and repetitively creating and terminating threads.

6.1.3 Request Management (RM) Layer

The user service interface module is responsible for retrieving user requests. It also handles any completion or error event from the lower layers, and maintains all data transfer sessions. On the other hand, it is in charge of communicating with users to report the progress of data transfers, performance data, and errors.

Upon receiving user requests from the request interface, this module decomposes them into one or multiple data transfer tasks. Figure 6-3 gives an example. Each storage device is associated with a task waiting queue for all tasks of reading/writing data from/into it. A user request is passed from the upper module and decomposed by the task composition and scheduling module, and then is regrouped into tasks according to the data location. Finally, the resultant tasks are dispatched to the corresponding task queues. Task regrouping here parallelizes task handling across different storages while ensuring that the storage I/O access within each group is handled by locally bound threads.

6.1.4 Protocol and Event Processing (PEP) layer

Capacity-based resource negotiation protocol

To coordinate the resource allocation and data transfer between two entities, namely the data source and data sink, we design a communication protocol for setting up connections and negotiating resource assignments. An important objective here is to offer sufficient I/O resources to each task while avoiding any deadlock and also conserving resources. To serve a data transfer task, we devise four types of I/O threads, i.e. readers and senders

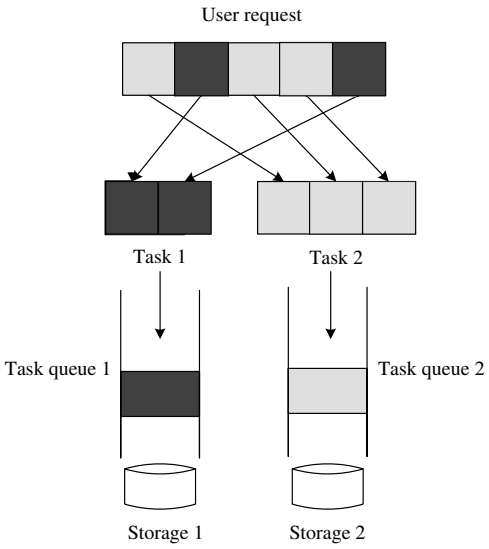


Figure 6-3: Task grouping mechanism. Different textures and colors represent requests over different devices

on the data source, and writers and receivers on the data sink. However, either the source or sink at each end point is unaware of the availability of threads at the other end. Therefore, a resource negotiation protocol is required to coordinate resource allocations at both ends.

Figure 6-4(a) illustrates the protocol of capacity-based resource negotiation. Herein, *Cap* is the bandwidth capacity of tasks or I/O devices. The source first determines the task capacity locally, and applies a certain number of I/O threads according to the ratio of the task's capacity and the device's physical capacity. Once the reader and sender threads are acquired, the source initiates a control channel to the data sink, and notifies the latter that the source is ready. The results of task capacity and the number of senders are sent to the sink for negotiating its resource assignments. The sink then computes the final task's capacity, and gets the I/O threads accordingly. Finally, after the sink sends back the confirmed data connections and task capacity, the source relinquishes any excessive senders and readers with the references to these confirmed parameters. The size of data block, viz., the processing unit for disk access and network communi-

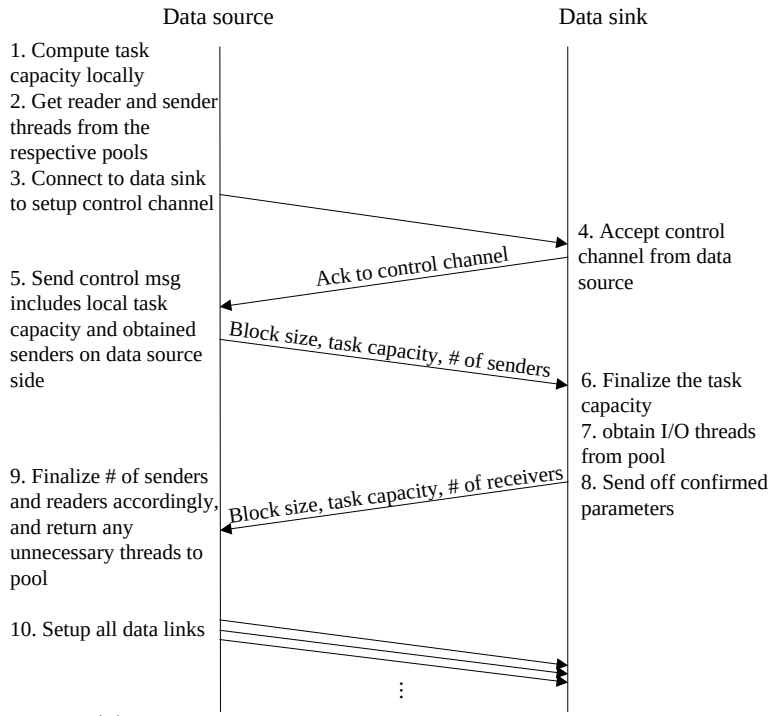
cation, is also negotiated during the aforementioned protocol handshakes. This mechanism guarantees each task has sufficient thread resources, while maximizing the number of concurrent tasks. After negotiating resources and assigning I/O threads, the data source initiates all data connections.

File metadata synchronization and payload transfer protocol

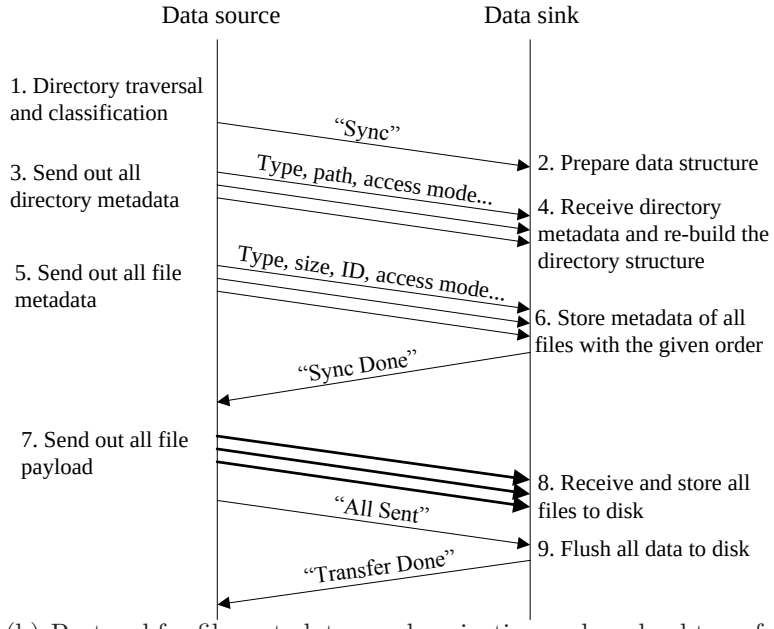
Metadata synchronization and payload transfer plays a vital role to the performance, especially while transferring massive numbers of files in wide-area networks. Small file transfers may cause the underlying transfer protocol not reaching the full network utilization due to short-duration transfers and connection start up/tear down overhead. Thereby, we design a three-step protocol in the PEP layer, as described in Figure 6-4(b). Herein, "file" refers to a regular file or a directory.

1) **File and directory scan.** The data source first scans all requested files and recursively traverses the involved directories using a Depth-First Search (DFS) algorithm. During this process, each file gets a file ID based on its sequence of being traversed. The file ID becomes a part of metadata and indexes the file among large transfer tasks. The metadata of regular files and directories will be saved to different data structures.

2) **Metadata synchronization and pre-processing.** As shown in Figure 6-4(b), the data source sends an "extend" control message to that data sink that starts to prepare all necessary data structures. Without waiting for any acknowledge message from the data sink, the source first sends all metadata of the file directories, and then sends all metadata of regular files, hard and symbolic links. The data sink takes different actions for different types of metadata, i.e., it simply creates the requested paths/directories upon the arrival of a directory, stores the metadata of files with the given order according to the file ID when file metadata arrives, and recreates the symbolic link to local files if it gets file links. Subse-



(a) Protocol for capacity-based resource negotiation



(b) Protocol for file metadata synchronization and payload transfer

Figure 6-4: Protocol design in PEP layer

quently, the whole directory structure is rebuilt at the data sink, and all metadata of regular files are stored and sorted based on file ID. This design ensures that before the actual data transfer begins, the target directory is

already in place. Finally, the data sink sends out a "Sync Done" message to the source, which marks the end of stage for synchronizing metadata. Therefore, instead of synchronizing one file per round trip time (RTT), the method commonly used and referred in Figure 1-3, our mechanism groups all records of file metadata together and streams them to the sink. Here, we only utilize only one RTT to signal the start and completion of the stage for synchronizing metadata. This significantly shortens the latency of synchronizing metadata and getting prepared for transferring file payloads.

3) **Pipelined payload transfer.** Afterwards, the data source sends out all files without any further exchange of control messages. The data sink can identify the incoming data block by the file ID in its header, and store it into the right file. After all files are sent, the data source and data sink finalize the data transfer by a single control message exchange. Similar to meta data synchronization, the proposed protocol here also groups all file payload transfer into a single stage, and only uses a single round trip to signal the completion of the entire task.

In summary, by dividing the data transfer process into stages and grouping the control messages in each stage, the proposed protocol for synchronizing file metadata and transferring file payloads greatly decreases the number of control message exchanges, and thereby shortens the total data transfer time. In addition, our design reuses the control and data links as much as possible among files, and avoids the overhead of establishing and tearing down connections for each file which are common in today's transfer software.

6.1.5 Data Access and Transmission (DAT) Layer

The buffer management module in the DAT layer uses a NUMA-aware bulk memory allocation. It first allocates a large trunk of memory for each task, and then partitions it into small buffers that are then allocated and

attached to a task that is ready for processing. The module determines the number of buffers for each task based on the number of assigned network threads. Meanwhile, by default, all buffers are pinned on to the NUMA node that directly connects the targeted network interface because TCP/IP incurs a more intensive protocol processing load than storage I/O. The buffers are also reused over the course of data transfer.

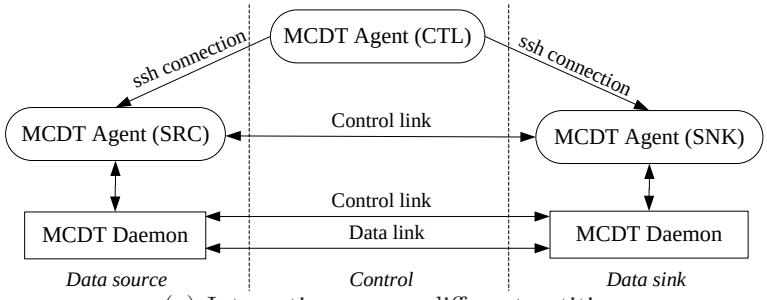
The two modules at the bottom of the DAT layer in Figure 6-2 are encapsulated by network I/O threads (senders/receivers) and storage I/O threads (readers/writers). At the data source, readers produce loaded buffers, and senders consume them and then return them to the list of free buffers. After the entire data transfer is completed, the I/O threads are returned to their respective pools for reuse, and a completion event is posted to the upper layer. The I/O threads are designed to be self-suspendable when waiting for tasks to minimize CPU consumption.

6.2 Implementation

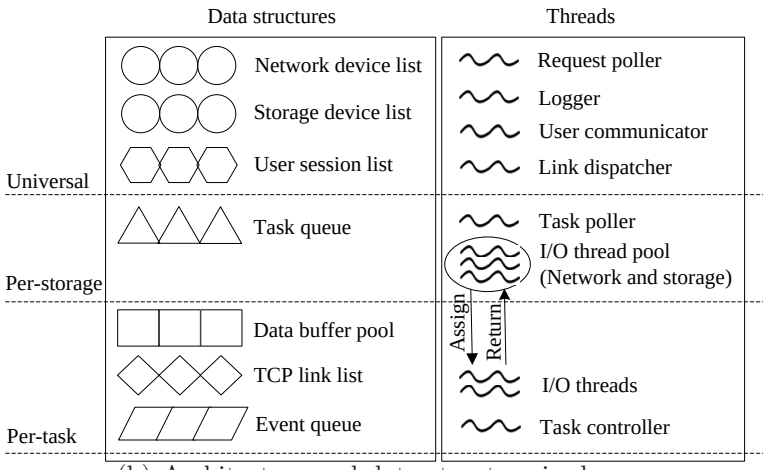
In this section, we describe the implementation of the reference implementation of the proposed framework, named Resource-Aware Asynchronous Data Replication with Multicore SYStem (RAMSYS). It comprises two components, a front-end agent and a background daemon, as shown in Figure 6-5(a). The agent is responsible for receiving the command line requests from users, relaying them to the daemon, and then reporting the progress/performance data or an error message during and after the data transfer. The entire software adopts a peer-to-peer design, and therefore naturally supports third-party data transfer.

The general process for handling a user request is as follows: firstly, RAMSYS accepts a user's command line request via the *Control (CTL)* entity that then initiates *ssh* connections and forwards requests to both

the data source and data sink hosts. Subsequently, the *sshd* servers at two end hosts launch *Data source (SRC)* and *Data sink (SNK)* agents (entities) that post the requests to their local daemon processes. The latters retrieve the request, and start to communicate with the remote peer.



(a) Interaction among different entities



(b) Architecture and data structure in daemon

Figure 6-5: RAMSYS implementation

6.2.1 Daemon Implementation

Figure 6-5(b) shows the multithreaded architecture and data structure of the background daemon process. It acts as the rendezvous point to collect system-wide requests for data transfer, manages I/O-related resources, and bootstraps data transfer tasks intelligently. There are three categories of data structures and threads in the daemon process. We detail them in the following categories.

Universal data structure and threads

This category of data structures and threads is shared globally and corresponds to the system-wide shared resources and entities. Specifically, the daemon process maintains a list of resource abstractions for all network and storage devices. The users' session list contains all user requests that currently are in processing, and each session element is created for one request. Among all running threads, the request poller retrieves a request from a local request pipe, and inserts a session element into the list of user sessions. Thereafter, it analyzes and partitions the request into several tasks, and then distributes them to different task waiting queues that correspond to different storage devices. Meanwhile, the link dispatcher listens on a well-known port, accepts incoming connections, and dispatches the link to the right task according to the first control message that comes through the link.

Per-storage data structure and threads

This category of data structure and threads is defined for I/O devices. Given the storage-centric design, task queues and their polling threads are created during system initialization and statically attached to their respective target storages. The universal request poller thread adds task to a task waiting queue of the target storage. Afterwards, the task poller thread of the storage activates a request by attaching a dedicated task controller thread and removes it from the task waiting queue. Furthermore, in our implementation, each storage and network device has both inbound and outbound data accesses (reads/writes), and thus, for a clean thread design, they have two separate thread pools for inbound and outbound access.

Per-task data structure and threads

This type of data structures and threads is assigned to newly arrived tasks. The task controller thread maintains a task's status, and manages its data structure and the associated threads during the life cycle of a task. We use a state transition diagram in Figure 6-6 to depict the life cycle of a data transfer task in a daemon process. At the beginning, the task is kept in the task waiting queue and in the "waiting" state. Once it is retrieved by a task polling thread, it acquires the "active" state, and the daemon creates and starts a task controller thread for the active task. The controller thread then makes reservations for thread resources from the corresponding I/O thread pool. If the target pool is empty, the target task enters the "suspend" state, and the task controller thread becomes inactive. Once the resource pool is not empty upon the completion of other tasks, the task controller thread then wakes up to query the pool again, and the corresponding task returns to the "active" state. After resource negotiation and assignment, the task controller thread obtains all the required resources, viz., assigned I/O threads. At this point, the task enters the "transferring" state. When the data transfer is completed, all assigned I/O threads post completion events to the task controller thread. The task then moves to the final "exit" state. After collecting all completed events, the task controller thread returns all I/O threads to the thread resource pools, wakes up any suspended task and updates the list of user sessions.

6.2.2 Optimizations

Instead of targeting a specific user case, the design objective of RAMSYS is to offer high-speed data transfer consistently under different workloads, and among various storage systems. RAMSYS is based on a highly par-

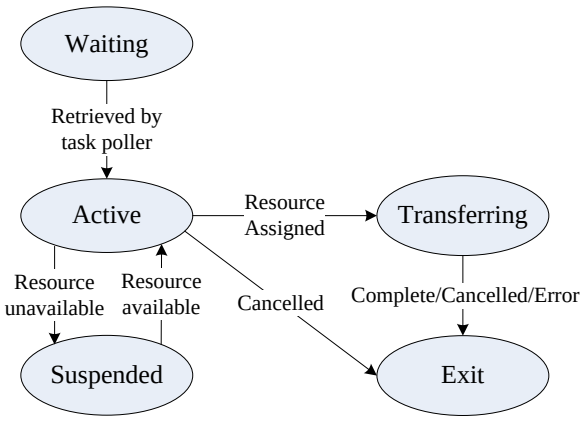


Figure 6-6: State transition diagram of data transfer task in RAMSYS

alleled framework, and integrates a variety of optimizations, wherein each of them works well for certain users. They must be applied in a unified fashion to boost the overall system I/O performance across different scenarios. We introduce the design and implementation of these optimizations in this section and validate their effectiveness in the subsequent experimental section.

Kernel-bypass storage I/O

The direct I/O operation offers kernel-bypass for accessing storage. By bypassing kernel page cache, it saves a copy between the kernel cache and user buffers, so avoids the overhead of context switch and related kernel management. This is particularly useful when using a large block size to access high-speed storages, such as SSD and network-based storage.

Adaptive I/O threads management

In RAMSYS, multiple I/O threads may be assigned to serve a single storage. Based on file sizes in a given task, these I/O threads will employ two different types of mechanism. The first one is the *stripping mode* for large files wherein all I/O threads will serve one file at a time, and each

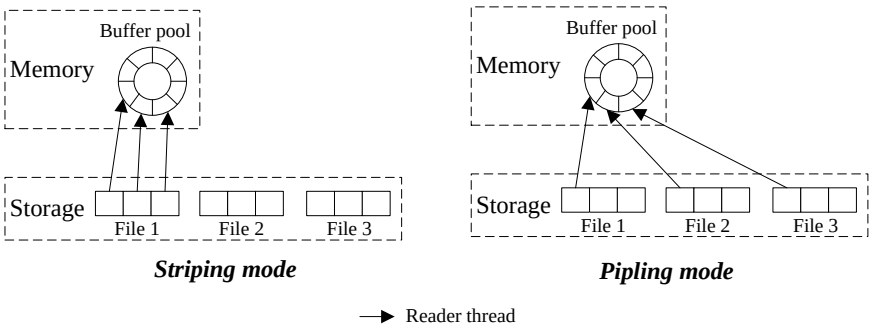


Figure 6-7: Different I/O multi-threading mode in RAMSYS

works on a fraction of the file. Figure 6-7 shows an example of assigning reader threads to transfer a number of files on data source host. Each file is segmented into three parts, each of which is handled concurrently by a separate reader thread (represented by the arrows in Figure 6-7). All files in the stripping model are handled one-by-one sequentially. This mode performs well while transfer large files, but it is inappropriate for a batch of small files because segmenting a small file confers no gain, but nevertheless incurs the overhead of coordinating all I/O threads for each file. The approach in this case is to allocate a single I/O thread for one file, and transfer multiple files in parallel with multiple threads. It is termed the *pipelining mode*. For example, in Figure 6-7, three reader threads work on three different files at the same time. After completing the transfer of one file, a thread moves on to the subsequent file in the queue. By adaptively switching between the pipelining and stripping modes, RAMSYS enhances system parallelism and thus increase the overall throughput.

Asynchronous non-blocking storage I/O support

RAMSYS also implements the asynchronous non-blocking data flow to enable parallelism at the data-block-level wherein a single storage I/O thread concurrently reads/writes multiple blocks of a given file. The heart of the RAMSYS non-blocking data flow rests upon our efficient storage AIO

module that supports the following two aspects:

1) The batch processing of multiple data blocks within a single function/system call minimizes the synchronization overheads to lock/release critical regions. Figure 6-8 demonstrates the state transition diagram of buffers and interactions among sender threads, reader threads, and the buffer pool at the data sender. To enable batch processing, the I/O thread will try to retrieve/post multiple buffers from/to buffer pools every time it acquires the buffer pool's lock, so as to minimize the synchronization overhead per-block.

2) An effective mechanism guarantees processing tasks at different pipeline stages in time, while avoiding unnecessary busy-waiting on any specific stage. Algorithm 2 details the design of the AIO access in RAMSYS. If it detects that no desired resource exists in the buffer pool, it then turns to check for any pending task in other stages. To avoid unnecessary non-blocking pollings, especially in large data blocks and low IOPS cases, the proposed data flow blocks on event processing and resource requesting when its current stage is the bottleneck of the entire processing flow, and chooses non-blocking processing when there are data/events to be consumed in the subsequent stage. This intelligent switch between blocking and non-blocking processing assures timely polling data buffers/events while avoiding unnecessary idle spins.

6.3 Experimental evaluation

This section first describes the configurations of the testbeds and workloads. We then evaluate and analyze each optimization in RAMSYS. At last, a comprehensive performance comparison is provided between RAMSYS and the other widely used data transfer tools.

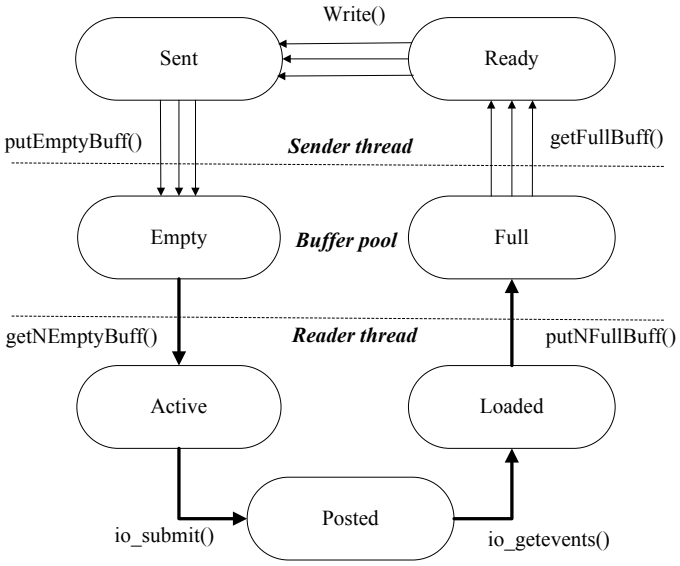


Figure 6-8: Block state transition of RAMSYS AIO module at data source

Algorithm 2: Data and event processing algorithm in the AIO module of RAMSYS

Input: BATCH SIZE: *batch*, TOTAL NUMBER OF BUFFERS: *total*

```
1  Number of available empty buffers: active ← 0 )
2  Number of buffers submitted to I/O context: posted ← 0
3  while have not transferred all the data do
4      if active + posted > total then
5          if active > 0orposted > batch then
6              Get active buffers from empty buffer pool (non-blocking)
7          else
8              Get active buffers from empty buffer pool (blocking)
9      while active > 0 and updated offset not reach the end-of-file do
10         Assign I/O offset to the first buffer in active buffer list
11         Submit the buffer to kernel via libaio API
12         Move the submitted buffer from active list to posted list
13         Update running offset
14     if posted > batch then
15         if posted = total then
16             Wait for completion events of the posted buffers (blocking)
17         else
18             Wait for completion events of the posted buffers
               (non-blocking)
19         Return all the completed buffers to the loaded buffer pool
20 return
```

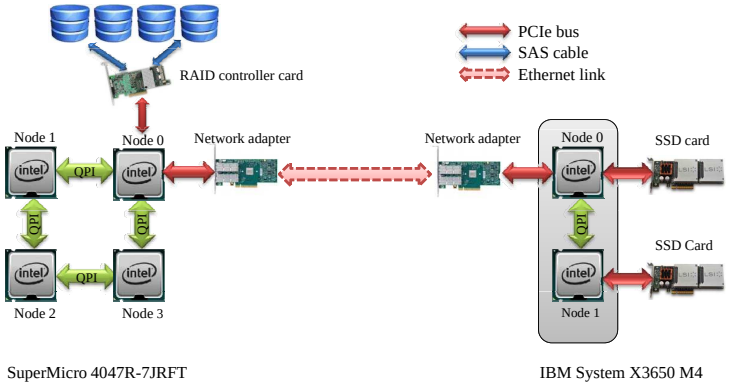


Figure 6-9: LAN testbed connectivity

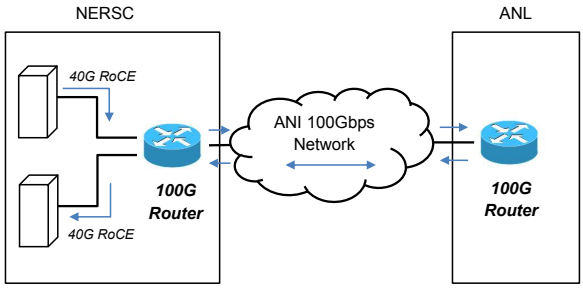


Figure 6-10: WAN testbed connectivity

6.3.1 Testbed and Workload Specifications

The local area network (LAN) testbed includes two state-of-the-art multicore systems, as demonstrated in Figure 6-9. Table 6.1 shows the hardware specifications. We configure the SuperMicro host with four RAID0 disks, each with eleven HDDs. All the disks are attached to the LSI RAID controller card on the motherboard. Meanwhile, the IBM system in the testbed includes two LSI Nytro WarpDrive cards, each connected to a separate NUMA node. Two hosts use a 40 Gbps Ethernet adapter for their network connections. PCI Gen3 connects all I/O devices with CPU nodes. Both servers run CentOS 6.5 with the 2.6.32-431 Linux kernel.

The wide area network (WAN) testbed belongs to DOE's ESnet (Energy Science Network). It provides a 40 Gbps long-haul link that extends 4,000 miles, from the National Energy Research Scientific Computing Center

Table 6.1: Server specifications - LAN

Motherboard	SuperMicro 4047R (LAN)	IBM System 3650 X3
Processor	Intel Xeon E5-4620 @ 2.60 GHz	Intel Xeon E5-2660 @ 2.20 GHz
CPU cores	4×8	2×8
Memory	4×80 Gbytes	2×64 Gbytes
Storage	4 × RAID0 HDDs	2 × LSI SSDs

Table 6.2: Server specifications - WAN

Motherboard	SuperMicro X10DRi	SuperMicro X10DRi
Processor	Intel Haswell Xeon E5-2643 @ 3.4 GHz	Intel Haswell Xeon E5-2643 @ 3.4 GHz
CPU cores	2×8	2×8
Memory	2×64 Gbytes	2×64 Gbytes
Storage	RAID0 HDDs	RAID0 Samsung SSDs

(NERSC) in Oakland, CA, to Argonne National Laboratory (ANL) near Chicago, IL and then loop-back to the NERSC (Figure 6-10). The two hosts in the testbed are located at the NERSC, and have a special loop-back configuration via ANL's 100 Gbps router to ensure a long network latency between them [6]. The minimum RTT between these them is 94.5 milliseconds. Table 6.2 lists their hardware configurations respectively. One host has 24 HDDs, grouped as a single RAID0 disk. The other host has 12 SSDs that are configured as a RAID0 disk, and it is used as the data source during our WAN evaluation.

We use three types of workloads. 1) Transfer of large bulk data: we fill the SSD disks with a few large files with sizes ranging from 50 to 200 Gbytes, and transfer them to the HDD disk array at the destination. 2) Transfer of massive numbers of small files: we downloaded ten different versions of Linux kernel source packages from its official site, and extracted their content and directory hierarchies to the SSD disks. The total data contain 440,004 files in 28,285 directories, and its total size is 5.8 Gbytes. Their detailed description is in Table 6.3. 3) Mixed workloads: we created 1,000

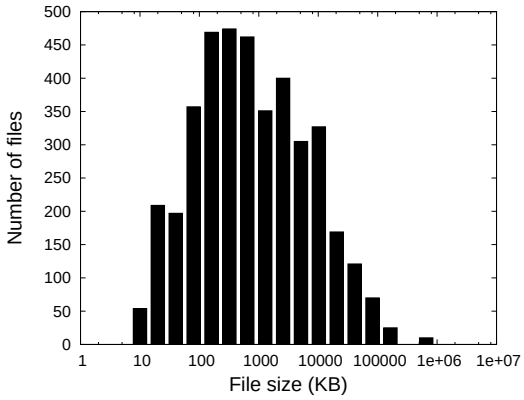


Figure 6-11: The distribution of file sizes for mixed workloads

Table 6.3: Linux kernel source file description in small file workload

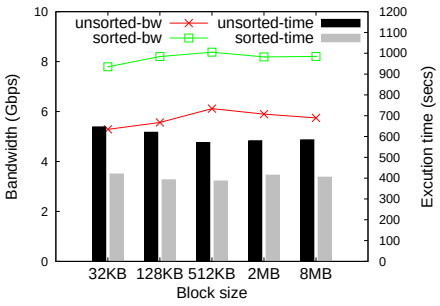
Version	Files	Directories	Total size (MB)	Avg size (KB)
2.6.32.67	30494	1878	416	13.9
3.2.71	37625	2345	502	13.6
3.4.108	38573	2389	515	13.7
3.12.47	44599	2871	610	13.6
3.14.52	45942	2948	610	13.6
3.18.21	47984	3078	633	13.5
4.1.7	49438	3207	650	13.5
4.2	50781	3376	685	13.8
4.3-rc1	51545	3439	696	13.8

files with different sizes in each RAID0 SSD/HDD disk array, and sent them over the 40G links to the disk array on the other host. The size of the files in the mixed workload follows a log-normal distribution, as shown by the histogram in Figure 6-11. We used Ganglia [120] on the LAN testbed and Graphite [121] on the WAN testbed to record the overall bandwidth and the CPU utilization. In addition, we measured the execution time of each software via the "time" utility in Linux.

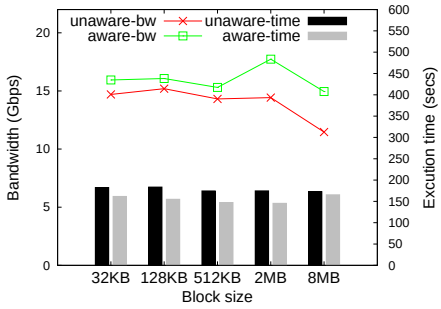
6.3.2 Evaluation of Proposed Optimizations

In this section, we quantitatively evaluate the different optimizations that are introduced in Section 6.2.2. The performance of RAMSYS are com-

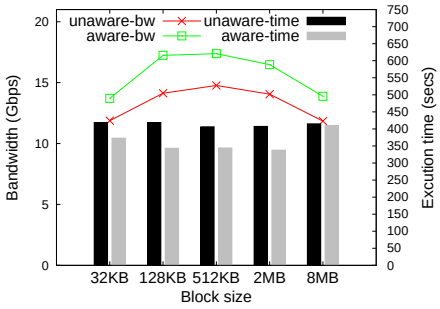
pared in two cases: with and without the optimizations, so as to understand the effectiveness of each one in different types of storage devices and scenarios. We use the large file and mixed workloads here because the small file workload largely depends on the protocol design rather than these optimizations.



(a) Sorted v.s. unsorted with mixed workload



(b) Aware v.s. unaware with bulk data



(c) Aware v.s. unaware with mixed workload

Figure 6-12: Comparisons of bandwidth and single-task latency between optimized and unoptimized RAMSYS

Effectiveness of file sorting

As discussed in Section 6.2.2, RAMSYS sorts all files in each queue according to the file's starting address. Figure 6-12(a) compares the sorted and unsorted transfers of 4000 files, i.e. the mixed workload, from a single RAID0 disk on LAN testbed. We observed an impressive improvement in bandwidth of 37% to 47% for the sorted transfers over the unsorted

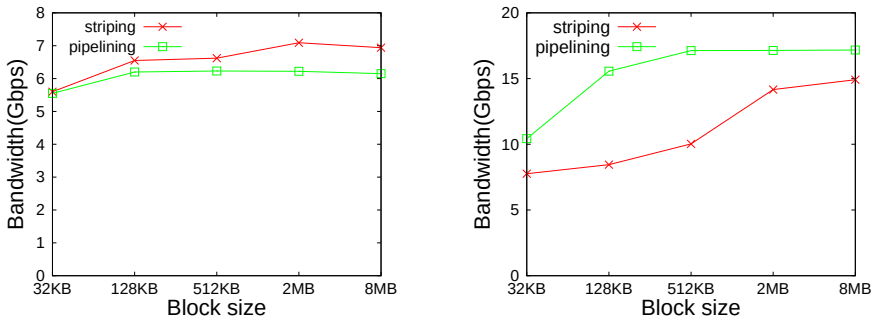
ones, and so confirmed the effectiveness of file sorting that can significantly reduce randomness for accessing storage.

Effectiveness of NUMA-awareness

NUMA-awareness is another critical optimization in RAMSYS. Figure 6-12 compares the results from two types of workloads: transfers of bulk data, and of mixed-size files. On average, the NUMA optimization delivers a 14.9% increase in bandwidth, and a 12.7% decrease in latency for bulk data transfer. For processing the mixed workload, we observed a 17.9% improvement in bandwidth and a 12.5% reduction in latency. Instead of depending on the default OS scheduling, RAMSYS utilizes its own pre-allocation and buffer management modules to pin I/O threads and their data to the NUMA node that connects directly with the I/O device involved. These results confirm the effectiveness of the thread-dependency-aware scheduling in RAMSYS across different workloads.

Effectiveness of multithreaded modes and file-level parallelism

In RAMSYS, multiple disk reader/writer threads can work in two multi-threading modes, striping and pipelining, as described in section 6.2.2. RAMSYS uses the striping mode for transferring large files, and pipelining for small file transfers. Herein we focus on the comparison of the two modes on the mixed workload. Figure 6-13 compares the bandwidths while accessing data from two different types of storage: the SSD tests read data from SSD, and send it to the memory of the other test host via the 40G LAN link, while the HDD tests read from HDD disks. Herein, we assign two reader threads to the HDD storage to compare the two disk I/O modes. The striping mode achieves a higher bandwidth in the HDD case than the pipelining mode, since the reader threads of the latter retrieve data of multiple files simultaneously, and increase the randomness of accessing



(a) HDD access (b) SSD access

Figure 6-13: Comparison of bandwidth between striping and pipelining modes in RAMSYS

HDD. Here, the sequence of accessing HDD plays a more important role than file-level parallelism wherein the striping mode preserves a better data sequence than the pipelining mode, and thus, attains a better performance. In contrary, SSD favors concurrent random accesses over sequential ones. Therefore, the pipelining mode has an 84% improvement in bandwidth compared to the striping mode. In summary, the appropriate application of each mode assures significant benefits in different cases.

Effectiveness of the AIO module

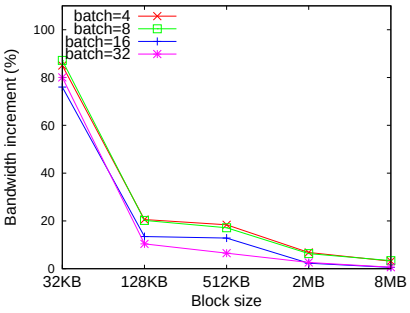
Figure 6-14(a) shows the bandwidth increment percentage of the AIO block-level parallelism over the synchronous thread-level parallelism for the LAN bulk data. "Batch size" here means that a thread calls Linux AIO system call to submit a specific number of block requests before it moves on to wait for completion, i.e. the storage I/O thread waits until the number of ready data buffers reaches the batch size, or it reaches the end of the task, and then processes them with only one system call. AIO outperforms the synchronous one, especially for a small block size. Its advantages are summarized the following: 1) The AIO module performs the batch processing of multiple data blocks with a single system call, and thus reduces the per-block cost. 2) More importantly, it only uses a single thread, while

the synchronous module utilizes as many as 16 parallel threads in this test. Therefore, it saves the resource required by multi-threading. 3) The AIO module also reduces the cost of synchronization over critical regions, since it batches multiple blocks into a single processing unit, and acquires the buffer pool lock only once. Figure 6-14(b) depicts the percentage of reductions in buffer pool lockings/synchronizations by an AIO module compared with those of a synchronous one. A larger batch size leads to more reductions in locking. However, it also adds more blocking time because a thread has to wait for a large number of blocks accumulated for consumption by a single buffer pool operation, and so can potentially lower the performance. Another noticeable observation is that the advantage of AIO diminishes when block size increases. That is because increasing block size reduces the total number of I/Os, and the cost per-bytes for synchronous access also declines, while asynchronous I/O adds waiting time for multiple large data blocks to be produced and consumed.

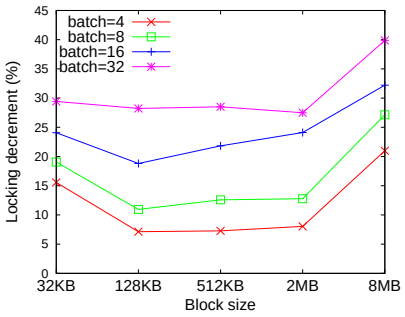
In addition, the AIO also benefits mixed workloads. Figure 6-14(c) compares the bandwidth of the AIO and synchronous disk accesses in the LAN tests. The AIO module demonstrates a stable performance across different block sizes, and there is a noticeable performance margin of between 15% and 110% over the synchronous I/O module.

6.3.3 Comparative Evaluation with Other Tools

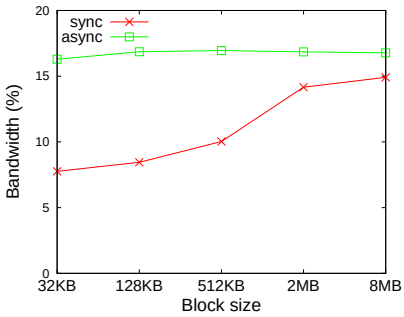
This section compares the performance of RAMSYS with three popular data transfer systems used by the high performance computing community, *GridFTP 5.2.5*, *BBCP 12.08.17.00.0*, and Aspera 3.6.6.112346. To ensure the best performance for GridFTP, we enabled its thread option, disabled all authentication operations and utilized its extended block mode (MODE E) for all data transfers [122]. For the BBCP, we also used its available options to avoid the overheads of checking DNS and the storage space at



(a) Bandwidth increment of bulk data



(b) Locking decrement of bulk data

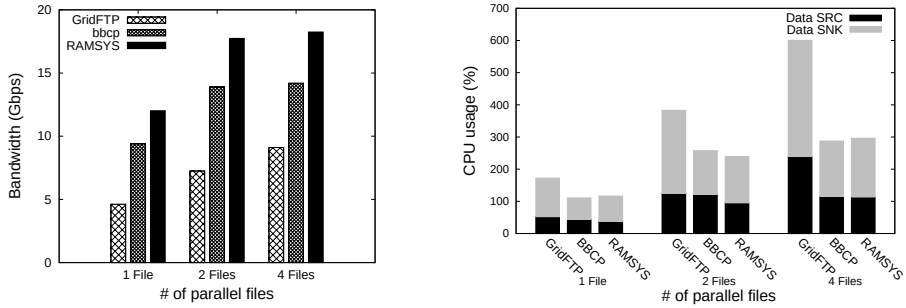


(c) Bandwidth of mixed workload

Figure 6-14: Bandwidth and locking of AIO disk access module compared with synchronous I/O module in RAMSYS over LAN testbed

the data sink. In all tests, both GridFTP and BBCP used 16 parallel TCP streams, viz., the best case that we observe on our testbeds. For Aspera, we observed only about 1.5 Gbps on LAN testbed and 4 Gbps on WAN testbed in bulk data and mixed workload tests. This is much less than the other three tools, and thus is ignored here.

1) **Bulk data transfer:** we process the bulk data workload by different numbers of parallel file transfers on the LAN testbed. Each concurrent file goes to a separate RAID0 disk on a SuperMicro server, resulting in a maximum of four concurrent files. During the parallel transfers of multiple files, we launched multiple instances of GridFTP and BBCP, one for each file, so to assure concurrency. On the other hand, in all test cases, we only needed to create a single RAMSYS daemon process to transfer all the data. Through thorough experiments, we observed that all of three software sys-

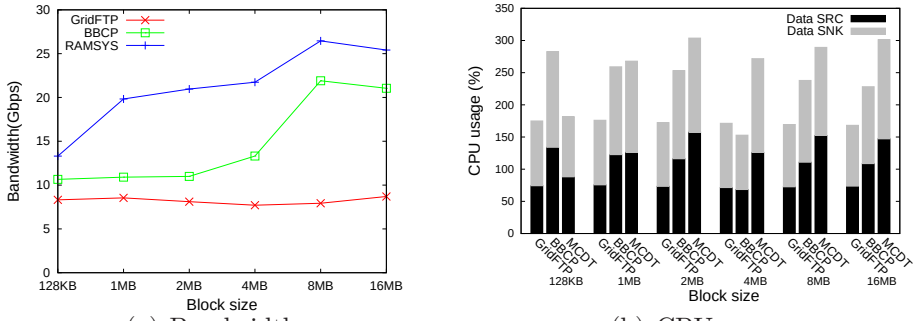


(a) Bandwidth (b) CPU usage

Figure 6-15: Comparison of bandwidth (a) and CPU usage (b) on bulk data transfers over LAN testbed

tems perform well at 2M block size, as shown in Figure 6-12, and therefore, we set a block size of 2M for the comparison studies. Figure 6-15 shows that RAMSYS performs the best in all tests, and scales well when the number of parallel file transfers increases. For more than two parallel files, RAMSYS's bandwidth exceeds 18 Gbps, which is very close to the bandwidth limit of data write to the LSI RAID controller in the SuperMicro host. Both RAMSYS and BBCP outperform GridFTP and use fewer CPU cycles because they utilize direct I/O operations, and also employ separate threads for storage and network I/O operations. Furthermore, RAMSYS also achieves 28% more bandwidth than does BBCP while it still retains a similar CPU load because it takes advantage of the multithreaded storage I/O for accessing SSD.

We also verify the advantages of RAMSYS in the bulk data transfer using the WAN testbed with different block sizes. During the tests, data are read from the SSD array and sent to the HDD array on the other host via the 40 Gbps long-haul link. Figure 6-16 shows the bandwidth and CPU usage of a single large file transfer. RAMSYS still significantly outperforms the other two in terms of bandwidth, confirming that the RAMSYS framework and its optimizations are scalable for long-distance data transfers.



(a) Bandwidth (b) CPU usage

Figure 6-16: Comparison of bandwidth (a) and CPU usage (b) on bulk data transfers over WAN testbed

2) **Massive small file transfer:** In this test case, we used Linux kernel source files, to compare the execution time of the four different transfer tools in preserving the tree structure of the Linux kernel while relocating all files. GridFTP supports multiple concurrent FTP sessions to transfer files in a directory, and here, we used eight concurrent FTP sessions. On the other hand, RAMSYS, BBCP and Aspera only used a single transfer session. We first test one kernel source tree of the kernel 3.18.21 that contains 47,984 files in 3,078 directories. Tables 6.4 and 6.5 compare the execution time on the LAN and WAN testbeds respectively. The "Ratio" values in the tables refer to the speed-up factor of RAMSYS over each of the comparative tools. BBCP performs the worst here due to its inefficient protocol design described in Section 1.1.2. GridFTP alleviates this drawback by dividing the workloads among multiple FTP sessions, and also enables the pipeline mode to transfer multiple files concurrently for each session. In addition, the multi-staged file processing protocol described in Section 6.1.4 affords RAMSYS a highly efficient mechanism for transferring a massive number of small files within a single FTP session. Not only does it deliver a 1.7x to 1235.9x speed-up compared to all alternatives in the LAN tests, but also demonstrates an even bigger improvement over BBCP and GridFTP for the long-haul WAN tests. Here, BBCP needs very long

Table 6.4: Execution time of transferring Linux kernel files over LAN testbed

	Single kernel directory		Ten kernel directories	
Software	Time (seconds)	Ratio	Time (seconds)	Ratio
BBCP	1177	130.8	88982	1235.9
GridFTP	27	3.0	246	3.4
Aspera	16	1.7	134	1.9
RAMSYS	9	1	72	1

time, of about several days, to complete the transfer of ten kernel trees, and we can not hold the shared WAN testbed to measure BBCP and have to mark "N.A" for BBCP in Table /reftab:ramsys-small-file-wan.

We also observe that it takes RAMSYS four seconds longer than Aspera did to transfer a single Linux directory in WAN testbed. On the other hand, for transferring ten Linux directories, RAMSYS achieve 3.9x speed-up over Aspera. Figure 6-17 shows the execution time of the two software tools for transferring different numbers of Linux kernel directories over the WAN testbed. For less than four directories, RAMSYS spends a large fraction of running time on authentication with ssh commandline scripts while Aspera utilizes the more efficient OpenSSL library for authentication. When the number of files increases, the file transfer time becomes dominant, and RAMSYS outperforms Aspera. In addition, in the case of eight directories or more, Aspera spends extra time to exit its program and significantly increases its total execution time.

All these comparisons confirm the efficiency and scalability of the file processing protocol in RAMSYS.

3) **Mixed workload transfer:** To evaluate the capability to handle a large number of mixed-size files, we generate 4,000 files with their sizes following a log-normal distribution, load them to each of the four RAID0 disks on the SuperMicro host, viz, $16,000$ files in total, and copy them to "/dev/null" of the IBM host. Again, for GridFTP and BBCP, we create

Table 6.5: Execution time of transferring Linux kernel files over WAN testbed

	Single kernel directory		Ten kernel directories	
Software	Time (seconds)	Ratio	Time (seconds)	Ratio
BBCP	5738	358.6	N.A.	N.A.
GridFTP	1136	71	11123	444.92
Aspera	12	0.8	98	3.9
RAMSYS	16	1	25	1

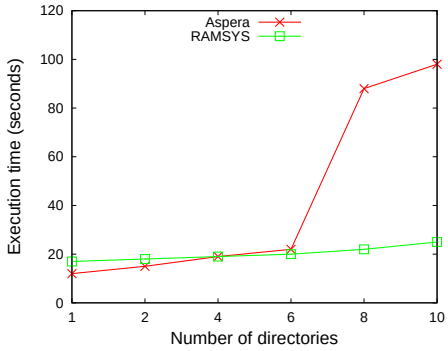
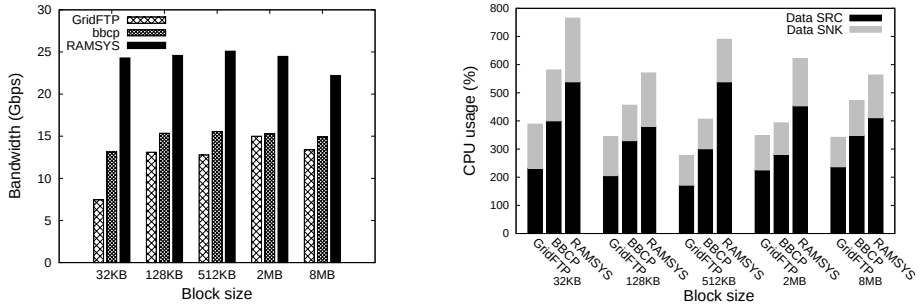


Figure 6-17: Execution time of Aspera and RAMSYS while transferring different number of kernel directories over WAN testbed

four concurrent instances, one for each RAID0 disk. Figure 6-18 compares their overall bandwidth performance and CPU usage with different block sizes. GridFTP only utilizes 4 CPU cores at most, and thus its bandwidth performance is lower than that of BBCP and RAMSYS. The CPU usage on the data source host is higher than that on the data sink host because this test case involves no disk write at the data sink. RAMSYS obtains consistently a higher bandwidth performance than do the other two, i.e., a 107.7% higher than GridFTP, and 63.2% higher than BBCP on average, while, in the meantime, with a higher CPU usage. This proves that RAMSYS has the best CPU scalability among the three transfer tool, and can fully utilize the multicore and I/O resources. In particular, when the block is small (32KB), RAMSYS demonstrates a significantly better capability of handling a large number of tasks and data blocks than the other two.

Figure 6-19 compares the bandwidth of three tools and their CPU us-



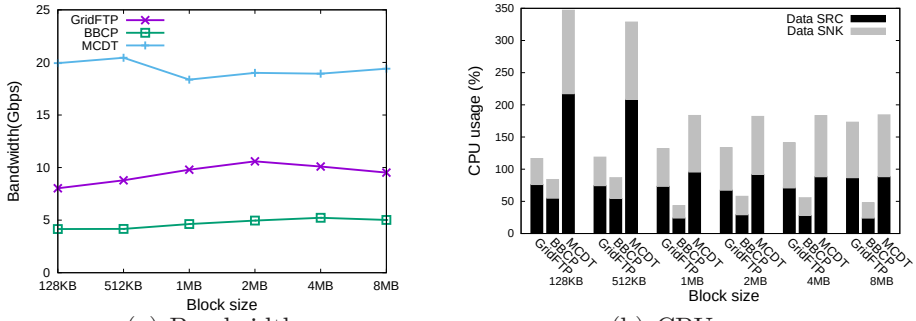
(a) Bandwidth (b) CPU usage

Figure 6-18: Comparison of bandwidth and CPU usage on transferring mixed workload over LAN

age for processing mixed workloads over the WAN testbed. RAMSYS is 2-3 times faster the other two tools. The BBCP protocol needs to synchronize file metadata one by one between the data source and data sink, and suffers more from a long RTT compared to the other two, which greatly compromises its overall performance.

Another interesting observation is that 512 KB appears to be the "sweet spot" and is favored in all mixed workload cases, as illustrated in Figures 6-12(a), 6-12(c), and 6-19(a). The reason is that when the block size is small, there are more data blocks to process, and each block incurs a constant amount of overhead, thereby RAMSYS incurs more aggregated overheads. On the other hand, when the block size is too big (for example, greater than 2 MB), most files in the mixed workload cannot even fill up a single block. A large portion of a data block is wasted, which also introduces unnecessary overheads. Furthermore, as depicted in Figure 6-11, the size of a large portion of mixed files is around 512 KB. For these reasons, a block size of 512 KB is our best practice in mixed file transfer.

To evaluate each individual optimization strategy employed in RAMSYS and quantify its gain in performance, we analyze all previous results, and undertake more experimental comparisons, e.g., RAMSYS without direct I/Os, versus GridFTP. Table 6.6 presents the final results. We make



(a) Bandwidth (b) CPU usage

Figure 6-19: Comparison of bandwidth and CPU usage on mixed workload transfer over WAN

several observations: 1) For transferring mixed workloads, asynchronous processing provides significant advantages in performance, while it is less effective in the tests of bulk data transfers. The reason is that, in bulk data cases, we choose to transfer a single file per storage device, so to avoid the bottleneck in performance due to writing to HDD. On the other hand, mixed workload tests involve processing more requests and incur more events than do the bulk ones. So asynchronous processing is more effective than synchronous processing in this case. 2) In contrast to GridFTP and BBCP, RAMSYS utilizes multiple parallel reader threads for accessing data in SSD, and gains another 17.21% more bandwidth for transferring bulk data. 3) Direct I/O delivers a major gain in performance of transferring bulk data. It is especially effective for loading data from SSD with a large block size. 4) During the mixed workload tests, we do not involve disk writing, direct I/O nor multithreaded storage access. The remaining optimizations, viz., NUMA-awareness, asynchronous processing, and more importantly file sorting, are the major contributors to performance.

Table 6.6: Breakdown of bandwidth increments compared to GridFTP

Optimizations	Bulk data (%)	Mixed workload (%)
NUMA-awareness	35.76	17.92
Asynchronous processing	8.27	47.09
Multithreaded storage	17.21	0
Direct I/O	98.85	0
File sorting	0	42.71
Total improvement	160.09	107.73

6.4 Summary

This chapter describes the resource-aware high-speed data transfer software, RAMSYS, which uses a novel asynchronous framework design. The previous monolithic designs do not fit state-of-the-art hardware, such as multi-/many-core servers, 100 Gbps networks, and NVRAM or SSD based storages. We introduce a multi-staged end-to-end data transfer design, wherein each stage is fully resource-driven and implements a flexible number of components for predefined functions, such as storage I/O, network communication, and request handling. RAMSYS relies on the asynchronous paradigm to maximize the concurrency of components, and thereby offers improved scalability and resource utilization in modern multicore systems. Furthermore, the proposed framework is extensible to integrate various optimization techniques, and we have shown several of them. Our testbed experimental results have quantitatively justified the effectiveness of each component in RAMSYS and the optimization techniques. The combination of the asynchronous processing and the integrated optimizations ensures superior performance for various workloads, attaining a 1.3x to 2.6x speedup for transferring large and mixed files, and a 1.7x to 1235.9x speedup for transferring directory trees over three widely used tools.

Chapter 7

Conclusion and Future Work

High performance hardware, such as multi-/many-core servers, 100 Gbps networks, and NVRAM or SSD based storages, emerges as the catalysis component to large-scale data-intensive computing systems. The problem of understanding and scheduling multicore resource is one of the hottest and most challenging topics in the system research. This dissertation quantitatively exposes the ineffectiveness of the existing multicore characterization methods and the data replication tools, and then pinpoints the mismatches between the defective designs in software and the new capability in modern hardware. Based on our mathematical and empirical analysis, we propose the multicore affinity metric to measure the NUMA remote access penalty and implement experimental tools and a data transfer system to supporting high speed data replication. In this chapter, we first conclude the research contributions of the dissertation, and then propose the future work directions.

7.1 Conclusion

To utilize the computing capacity in a multi-core system, modern high-speed data replication applications usually adopt the multi-threaded de-

sign and multiplex multiple network interfaces to enhance concurrency and improve the aggregated performance. On the other head, Non-uniform memory access (NUMA) multicore system introduce another dimension of complexity (i.e., architectural complexity) that must be addressed in designing high performance applications. It remains a persistent challenge to efficiently utilize the abundant resources while offering superior performance to different user scenarios in multicore computers. It is crucial to understand and utilize applications I/O characteristics, thereby to make decisions with respect to multicore-aware resource allocation, assignment and optimization to minimize the data I/O overheads and to improve the overall system throughput. In this dissertation, we detail our new findings and solutions on characterizing and modeling the I/O performance in NUMA systems, attaining multicore-aware resource scheduling, designing resource-aware asynchronous data pipelines, and ensuring high performance over different workloads.

At first, to develop a comprehensive understanding to the characteristics of I/O access, we run bulk data transfer applications on a state-of-the-art host while collecting the counter readings of the I/O and multicore related hardware events. No existing study did this type of the quantification work. We fist look into the widely-used hardware events, for example, LLC misses and resource stalk cycles, and identify the high cost incurred by inter-node buses. Further analysis uncovers the critical impact of prefetch contention and cache coherency traffic on NUMA effects. Base on our new findings, a new metric is proposed to quantify the NUMA remote access penalty, and support dynamic NUMA-aware resource scheduling.

Furthermore, we design an empirical performance modeling method for NUMA-aware resource scheduling based on pre-profiling. We first demonstrate the ineffectiveness of the existing performance modeling solutions on the modern multicore platforms, and reveal the mismatches between

these solutions and current hardware. The dissertation then provides an intelligent I/O performance modeling method without even involving expensive I/O hardware and time-consuming I/O operations. The model is verified and confirmed by the actual I/O benchmarking results, including asynchronous disk I/O, TCP/IP network I/O and the RDMA network I/O. We present concrete examples to show that this model can be applied to reduce the tedious I/O characterization workload, predict the aggregate I/O performance and mitigate resource contentions.

Subsequently, the dissertation claims that the mathematical optimization problem of resource scheduling on a multi-core platforms remains widely open. Different hardware and workload characteristics, the impact of co-scheduled tasks, NUMA factors and dynamic system loads all contribute to its complexity. This dissertation provides the formal problem definition of the task scheduling for data replication, and represent it with a two-staged graph model. Under this definition, attaining a scheduling solution is then transformed to solving a min-sum-max resource allocation problem (MSMRAP) that include integer variables. We then show that it is a NP-complete problem that has a high computation complexity, even for its relaxed form. We propose a potential solution based on a divide and conquer algorithm. Its solution space still grows exponentially with the number of cores and the number of tasks to be co-scheduled. We thus aimed to solve the problem with an empirical solution. We implement a NUMA-aware thread and memory scheduling module in BBCP tool. The evaluation results with our local testbed show significant advantages of the optimized BBCP by significant throughput improvements over the original implementation, i.e. 10.83% to 220% in the memory-based tests and 6.45% to 14.3% in the Storage Area Network tests.

Nevertheless, NUMA-awareness is only one aspect of the multicore optimization in data replication applications. More integrated design strate-

gies, such as multi-threading, kernel-bypass, asynchrony, and event-driven techniques are necessary to ensure high parallelism and performance over different workloads. This dissertation details a complete high-speed data transfer solution, named Resource-Aware Asynchronous Data Replication with Multicore SYStem (RAMSYS). We first systematically introduce its framework design by a layered structure and a staged task processing pipeline. Each stage is assigned with dedicated threads which are preallocated via the hardware-characteristic-aware initialization module. RAMSYS affinitizes its threads and their memory space with designated cores and reuses them among multiple file transfers to guarantee NUMA-awareness and minimize resource allocation overhead. Meanwhile, we regroup the user requests with a storage-centric design, and link the processing stages via an event-driven driven mechanism to maximize system concurrency. we also propose a capacity-based resource negotiation protocol, and a file metadata synchronization and payload transfer protocol to optimize the resource scheduling process. In addition, the software implementation also integrates various I/O related optimizations, including file-level sorting, block-level asynchronism, direct I/O and adaptive multi-threading modes, to further improve performance across different workloads.

Finally, to confirm the effectiveness of the proposed solution in real word use cases, we deploy RAMSYS system to our local data center, and the nation-wide Department of Energy (DOE) networks. Different representative workloads with large, small and random sized data files are also generated over different storage systems to test the performance of various user scenarios. We first prove the benefit of each optimization included in RAMSYS by comparing the performance before and after enabling it. We then run other well-known data replication tools, GridFTP, BBCP and Aspera, on the same workloads and testbeds to compare with RAMSYS software. These comprehensive experimental studies demonstrates the un-

rivaled advantages of RAMSYS, including 1.3x to 2.6x speedup on large, small and mixed file transfer, and 3x to 1236x speedup directory tree transfer. The significant performance gain results from the joint effect of high efficient software framework and customized optimizations to each user case.

7.2 Future Works

Inspired by the work and research outcomes in this dissertation, the future works focus on improving the NUMA-aware scheduling and harnessing the models in this dissertation in real application designs to further enhance the performance and user experience of sharing data among science, industry, and end-consumers, in particular, to integrate the NUMA I/O performance model into the resource affinity configuration setup in RAMSYS and other data-intensive programs and to apply the proposed NUMA scheduling factor to support real-time resource scheduling.

7.2.1 NUMA-aware thread and memory migration

A brute-force NUMA-aware scheduling is to bind all relevant threads into the same processor. However this approach will incur unbalanced workloads and resource contention on local resources, and creates a hot spot in system. Consequently, the penalty of resource contention potentially overwhelms the benefit of NUMA-awareness. The resource scheduler on NUMA multicore platform must make a trade-off between NUMA-awareness and contention-awareness in Section 5.2.3, and make adaptive decision to migrate some local tasks off to remote nodes. One future work targets to design a balanced scheduling among locality, contention and fairness. Existing state-of-the-art contention-aware algorithms for NUMA systems work as follows: they identify the target threads that share the same memo-

ry/CPU domain and in the meantime interfere each other's performance, and then choose and migrate part of the target threads to a different domain. This approach,referred as "NUMA-agnostic migration", may create a situation where a running thread access its allocated memory that is located in a different domain from its own. On the other hand, how to avoid unnecessary migration is another aspect of the scheduling problem. Frequent migration of threads is expensive because the associated CPU overhead is high and more importantly, the cache affinity can be not preserved. One improvement is to consider a thread and its memory together and migrate both to avoid the remote accesses and overloaded memory controllers. Meanwhile, the target selection for migration must take the size of memory into consideration because memory migration is extremely costly. The overall strategy is to minimize such a operation as many as possible and to avoid the repetitive migrations between the same targets.

7.2.2 Interrupt affinity control

In this dissertation, we detailed the effects of controlling the thread and memory affinity on NUMA platform and integrated this strategy in the read-world application. More aspects of resource affinity need further investigation for their potential use in real-world applications, for example, the techniques of controlling interrupt affinity, including Receive-side Scaling (RSS), Receive Packet Steering (RPS) and Receive Flow Steering (RFS). The problem is that all hardware interrupts end up getting serviced by CPU 0 by default in most Linux systems. This results in a system bottleneck and reduced performance. The interrupts must be distributed among CPUs to eliminate the system hot spot and in the meantime to preserve the access locality among the I/O path. Therefore it might be appropriate to distribute interrupts across cores. Modern network adapters have the capacity to support multiple interrupt queues, each of which can be served

by a separate CPU core. Packets enter the queues according to the hash value that is calculated from the pair of communicating IP addresses/ports. As a result, all packets for the same TCP session always end up in the same queue that has a dedicated interrupt handler. The multi-queue network device allows us to map interrupts to particular CPUs and cores so to spread out its load. Such a feature can be potentially incorporated into the data replication applications for the future 100Gbps/1Tbps network adapters.

7.2.3 Load balancing and work stealing among test queues

In RAMSYS, a thread is attached to a specific waiting queue associated with a storage device, and only serves the tasks from the queue. It is possible that one queue is overwhelmed by a large number of tasks and workload, while other queues and their serving threads are starving for tasks. This occurs especially when the system need to on-line process heterogeneous dynamic workloads. To achieve scalable performance, data intensive applications often need a real-time dynamic mechanism to balance loads among multiple task queues. Work stealing is a popular choice of distributed dynamic load balancing, but its performance impact to large-scale multi-user systems is not well understood. The work stealing algorithm with high efficiency and low overhead is indispensable to fully utilize many-/multi-core systems for high system throughput given unbalanced workloads.

7.3 Summary

Resource scheduling on multicore platforms is challenging and also critical to data replication applications. An extensive list of factors, i.e., NUMA asymmetry, storage characteristics, device capability, multi-task parallelism

and load balancing, must be considered together to consistently achieve high performance for various real world data replication workloads and network systems. This dissertation first provides in-depth analysis to understand the behavior of the underlying state-of-the-art hardware, subsequently proposes appropriate performance modeling method and metrics to measure and evaluate target systems and design strategies, and finally integrate these contributions to a real-word software system to demonstrate the effectiveness and efficiency of our proposed optimizations and the derived NUMA-aware design for high-speed data replication. Furthermore, we comparatively evaluate and compare our RAMSYS with the current widely-used data replication tools. The test results reveal its superior performance over the other existing solutions. At last, we offer several future work directions to further enrich the multi-/many-core high performance computing research.

Bibliography

[1] D.T. Nukarapu, Bin Tang, Liqiang Wang, and Shiyong Lu. Data replication in data intensive scientific applications with performance guarantee. *Parallel and Distributed Systems, IEEE Transactions on*, 22(8):1299–1306, Aug 2011.

[2] Mahadev Satyanarayanan, P. Bahl, R Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, Oct 2009.

[3] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law through EPI throttling. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 298–309, May 2005.

[4] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, May 2005.

[5] IEEE P802.3ba 40 Gb/s and 100 Gb/s Ethernet Task Force. Available at "http://www.ieee802.org/3/ba/".

[6] B. Tierney, E. Kissel, M. Swany, and E. Pouyoul. Efficient data transfer protocols for big data. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–9, Oct 2012.

[7] Amazon.com Inc. AWS Import/Export Snowball. Available at https://aws.amazon.com/importexport/.

[8] Aspera online page. Available at http://asperasoft.com/.

[9] B.M. Tudor, Yong Meng Teo, and S. See. Understanding off-chip memory contention of parallel programs in multicore systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 602–611, Set 2011.

[10] L.L. Pilla, C.P. Ribeiro, D. Cordeiro, Chao Mei, A. Bhatele, P.O.A. Navaux, F. Broquedis, J. Mehaut, and L.V. Kale. A hierarchical approach for load balancing on parallel multi-core systems. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 118–127, Sept 2012.

[11] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multicore architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[12] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, pages 16–29, 2010.

[13] Osmin Dumitru, Ralph Koning, and Cees De Laat. 40 Gigabit Ethernet: Prototyping transparent end-to-end connectivity. In *The TERENA Networking Conference 2011 (TNC 2011)*, 2011.

[14] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 277–289, Santa Clara, CA, July 2015. USENIX Association.

[15] Abdullah Kayi, Edward Kornkven, Tarek El-Ghazawi, Samy Al-Bahra, and Gregory Newby. Performance evaluation of clusters with ccNUMA nodes - a case study. In *The 10th IEEE International Conference on High Performance Computing and Communications*, pages 320–327, Septempber 2008.

[16] Zoltan Majo and Thomas Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, pages 1–10, 2011.

[17] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011.

[18] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[19] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The globus

striped gridftp framework and server. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, 2005.

[20] Andy Hanushevsky. BBCP online page, 2012. Available at http://www.slac.stanford.edu/~abh/bbcp/.

[21] Yunhong Gu and Robert L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Comput. Netw.*, 51(7):1777–1799, May 2007.

[22] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, Brian Tierney, and Eric Pouyoul. Protocols for wide-area data-intensive applications: Design and performance issues. In *Supercomputing 2012*, November 2012.

[23] Xiao Zhang, Sandhya Dwarkadas, Girts Folkmanis, and Kai Shen. Processor hardware counter statistics as a first-class system resource. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 14:1–14:6, San Diego, CA, 2007.

[24] Douglas Thain and Christopher Moretti. Efficient access to many small files in a filesystem for grid computing. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 243–250, Washington, DC, USA, 2007. IEEE Computer Society.

[25] Jon Dugan. iperf benchmark, June 2011. Available at http://sourceforge.net/projects/iperf/.

[26] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[27] A. Ros, B. Cuesta, R. Fernandez-Pascual, M.E. Gomez, M.E. Acacio, A. Robles, J.M. Garcia, and J. Duato. Extending Magny-Cours cache coherence. *Computers, IEEE Transactions on*, 61(5):593–606, 2012.

[28] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Quantifying NUMA and contention effects in multi-GPU systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–7, 2011.

[29] Shoaib Akram, Manolis Marazkis, and Angelos Bilas. NUMA implications for storage I/O throughput in modern servers. In *3rd Workshop on Computer Architecture and Operating System co-design (CAOS'12)*, 2012.

[30] Cheng Li, I. Goiri, A. Bhattacharjee, R. Bianchini, and T.D. Nguyen. Quantifying and Improving I/O Predictability in Virtualized Systems. In *IEEE/ACM International Symposium on Quality of Service (IWQoS)*, pages 1–12, June 2013.

[31] Cheng Li, Philip Shilane, Fred Douglis, Darren Sawyer, and Hyong Shim. Assert(!Defined(Sequential I/O)). In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[32] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 501–512, Philadelphia, PA, June 2014. USENIX Association.

[33] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: A Container-based Flash Cache for Compound Objects. In *Proceedings of the 16th International Middleware Conference (ACM/IFIP/USENIX Middleware 15)*, pages 61–73. ACM, 2015.

[34] Cheng Li, Shihong Zou, and Lingwei Chu. Online Learning Based Internet Service Fault Diagnosis Using Active Probing. IEEE ICNSC, 2009.

[35] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. NUMA-aware algorithms: the case of data shuffling. In *The biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[36] Patrick McCormick, Ryan Karl Braithwaite, and Wuchun Feng. Empirical memory-access cost models in multicore NUMA architectures. In *2011 International Conference on Parallel Processing (ICPP 2011)*, January 2011.

[37] D.R. Kaeli, L. L. Fong, R. C. Booth, K. C. Imming, and J. P. Weigel. Performance analysis on a CC-NUMA prototype. *IBM Journal of Research and Development*, 41(3):205–214, 1997.

[38] Martin Schmollinger and Michael Kaufmann. kNUMA: A model for clusters of SMP-machines. In *Parallel Processing and Applied Mathematics*, pages 42–55. Springer Berlin Heidelberg, 2002.

[39] K.W. Cameron and Xian-He Sun. Quantifying locality effect in data access delay: memory logP. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003.

[40] M. Forsell. A PRAM-NUMA model of computation for addressing low-tlp workloads. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, 2010.

[41] Patrick McCormick, Ryan Karl Braithwaite, and Wuchun Feng. Empirical memory-access cost models in multicore NUMA architectures. Technical report, Los Alamos National Laboratory (LANL), January 2011.

[42] Ryan Karl Braithwaite, Wuchun Feng, and Patrick McCormick. Automatic NUMA characterization using Cbench. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, 2012.

[43] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego, CA, 2010.

[44] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using papi for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*, Urbana, Illinois, June 2001.

[45] Arnaldo Carvalho de Melo. Performance counters on linux the new tools. In *Linux Plumbers Conference*, September 2009.

[46] Stéphane Eranian. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, MSPC '08, pages 26–30, Seattle, Washington, 2008.

[47] Sergey Blagodurov and Alexandra Fedorova. User-level scheduling on NUMA multicore systems under linux. In *Linux Symposium 2011*, 2011.

[48] Stephen Ziemba, Gautam Upadhyaya, and Vijay S. Pai. Analyzing the effectiveness of multicore scheduling using performance counters. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[49] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[50] Viren Kumar and James Delgrande. Optimal multicore scheduling: An application of asp techniques. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 604–609, Berlin, Heidelberg, 2009. Springer-Verlag.

[51] N. Mastronarde, K. Kanoun, D. Atienza, P. Frossard, and M. van der Schaar. Markov decision process based energy-efficient on-line scheduling for slice-parallel video decoders on multicore systems. *Multimedia, IEEE Transactions on*, 15(2):268–278, Feb 2013.

[52] Jihye Kwon, Kang-Wook Kim, Sangyoun Paik, Jihwa Lee, and Chang-Gun Lee. Multicore scheduling of parallel real-time tasks with multiple parallelization options. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 232–244, April 2015.

[53] Shekhar Srikantaiah, Reetuparna Das, Asit Mishra, Chita Das, and Mahmut Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[54] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[55] Yuejian Xie and Gabriel H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 174–183, Austin, TX, USA, 2009.

[56] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, Portland, Oregon, 2009.

[57] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 89–102, Nuremberg, Germany, 2009.

[58] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch-scheduling: Enhancing both performance and fairness of shared dram systems. In *35th International Symposium on Computer Architecture*, 2008.

[59] E. Ipek, O. Mutlu, J.F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 39–50, June 2008.

[60] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 18:1–18:18, Boston, MA, 2007.

[61] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 146–160, 2007.

[62] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multicore architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[63] B. Goglin and N. Furmento. Enabling high-performance memory migration for multithreaded applications on Linux. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–9, 2009.

[64] R. Knauerhase, P. Brett, B. Hohlt, Tong Li, and S. Hahn. Using os observations to improve performance in multicore systems. *Micro, IEEE*, 28(3):54–66, May 2008.

[65] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 153–166, Paris, France, 2010.

[66] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, Pittsburgh, Pennsylvania, USA, 2010.

[67] Mohammad Banikazemi, Dan Poff, and Bulent Abali. Pam: A novel performance/power aware meta-scheduler for multi-core systems. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 39:1–39:12, Austin, Texas, 2008.

[68] R.L. McGregor, C.D. Antonopoulos, and D.S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 28a–28a, April 2005.

[69] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45, 2013.

[70] Boost application performance using asynchronous I/O. Available at http://www.ibm.com/developerworks/library/l-async/.

[71] Kernel asynchronous I/O (AIO) support for Linux. Available at http://lse.sourceforge.net/io/aio.html.

[72] T. Ito, H. Ohsaki, and M. Imase. On parameter tuning of data transfer protocol gridftp for wide-area grid computing. In *Broadband Networks, 2005. BroadNets 2005. 2nd International Conference on*, pages 1338–1344 Vol. 2, Oct 2005.

[73] T. Ito, H. Ohsaki, and M. Imase. Gridftp-apt: automatic parallelism tuning mechanism for data transfer protocol gridftp. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 8 pp.–461, May 2006.

[74] T. Ito, H. Ohsaki, and M. Imase. Automatic parameter configuration mechanism for data transfer protocol gridftp. In *Applications and the Internet, 2006. SAINT 2006. International Symposium on*, pages 7 pp.–38, Jan 2006.

[75] G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, and J. Saltz. A dynamic scheduling approach for coordinated wide-area data transfers using gridftp. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008.

[76] Rajkumar Kettimuthu, Gayane Vardoyan, Gagan Agrawal, P. Sadayappan, and Ian Foster. An elegant sufficiency: Load-aware differentiated scheduling of data transfers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 46:1–46:12, 2015.

[77] Che Huang, Chawanat Nakasan, Kohei Ichikawa, and Hajimu Iida. A multipath controller for accelerating gridftp transfer over sdn. In *e-Science (e-Science), 2015 IEEE 11th International Conference on*, pages 439–447, Aug 2015.

[78] K. Chard, S. Tuecke, and I. Foster. Efficient and secure transfer, synchronization, and sharing of big data. *Cloud Computing, IEEE*, 1(3):46–55, Sept 2014.

[79] Andrew Hanushevsky, Artem Trunov, and Les Cottrell. Peer-to-peer computing for secure high performance data copying. In *In Proc. of the 2001 Int. Conf. on Computing in High Energy and Nuclear Physics (CHEP 2001), Beijng*, 2001.

[80] I. Gorton, A. Wynne, J. Almquist, and J. Chatterton. The medici integration framework: A platform for high performance data streaming applications. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, pages 95–104, Feb 2008.

[81] I. Gorton, Zhenyu Huang, Yousu Chen, B. Kalahar, Shuangshuang Jin, D. Chavarria-Miranda, D. Baxter, and J. Feo. A high-performance hybrid computing approach to massive contingency analysis in the power grid. In *e-Science, 2009. e-Science '09. Fifth IEEE International Conference on*, pages 277–283, Dec 2009.

[82] S. Bradley, F. Burstein, B. Gibbard, and D. Katramatos. Terapaths: a qos-enabled collaborative data sharing infra-structure for peta-scale computing research, computing. In *High Energy and Nuclear Physics (CHEP)*, 2006.

[83] Zdenek Maxa, Badar Ahmed, Dorian Kcira, Iosif Legrand, Azher Mughal, Michael Thomas, and Ramiro Voicu. Powering physics data transfers with fdt. *Journal of Physics: Conference Series*, 2011.

[84] Yunhong Gu and Robert L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Computer Networks*, 51(7):1777 – 1799, 2007.

[85] John Bresnahan, Michael Link, Rajkumar Kettimuthu, and Ian Foster. Udt as an alternative transport protocol for gridftp. In *International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*, May 2009.

[86] Aspera, an IBM company. Aspera fasp high speed transport. *Technical Whitepaper*, 2015.

[87] B. Eckart, Xubin He, and Qishi Wu. Performance adaptive udp for high-speed bulk data transfer over dedicated links. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10, April 2008.

[88] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. *A remote direct memory access protocol specification. RFC 5040*, October 2007.

[89] Ping Lai, H. Subramoni, S. Narravula, A. Mamidala, and D.K. Panda. Designing efficient ftp mechanisms for high performance data-transfer over infiniband. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 156–163, September 2009.

[90] H. Subramoni, Ping Lai, R. Kettimuthu, and D.K. Panda. High performance data transfer in grid environment using gridftp over infiniband. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 557–564, May 2010.

[91] Mellanox Technologies Inc. *Performance Tuning Guidelines for Mellanox Network Adapters*, July 2012.

[92] InfiniBand Trade Association. InfiniBand architecture specification. *Release 1.2.1*, 2006.

[93] Dennis Dalessandro, Ananth Devulapalli, and Pete Wyckoff. iSER storage target for object-based storage devices. In *The Fourth International Workshop on Storage Network Architecture and Parallel I/Os*, September 2007.

[94] Andi Kleen. libnuma/numactl and numa api for 2.6 released, 2004. Available at "http://lwn.net/Articles/67005/".

[95] Stephanie Moreaud and Brice Goglin. Impact of NUMA effects on high-speed networking with multi-Opteron machines. In *The 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 24–29, 2007.

[96] John McCalpin. Optimizing AMD Opteron memory bandwidth, part 1: Single-thread, read-only, 2010.

[97] John Levon and Philippe Elie. Oprofile, Auguest 2012. Available at http://oprofile.sourceforge.net/.

[98] Annie Foong, Jason Fung, Don Newell, Seth Abraham, Peggy Irelan, and Alex Lopez-Estrada. Architectural characterization of processor affinity in network processing. In *IEEE International Symposium*

on Performance Analysis of Systems and Software (ISPASS 2005), pages 207–218, 2005.

[99] David Kanter. Sandy bridge for servers. Technical report, Intel Corporation, July 2012.

[100] David Levinthal. Tutorial: Intel Core i7 and Intel Xeon 5500 microarchitecture, optimization and performance analysis. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2010.

[101] Nigel Griffiths. nmon for linux, June 2012. Available at http://nmon.sourceforge.net/pmwiki.php.

[102] Intel Corporation. *An Introduction to the Intel QuickPath Interconnect*, January 2009.

[103] David Kanter. Intel's sandy bridge microarchitecture. Technical report, Intel Corporation, September 2012.

[104] Intel Corporation. *Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3B*, March 2013.

[105] John Beckett. Memory performance guidelines for Dell PowerEdge 12th generation servers. Technical report, Dell Inc, July 2012.

[106] InfiniBand Trade Association. *InfiniBand Architecture Specification Release 1.2.1 Annex A16: RoCE*, April 2010.

[107] *AMD Opteron 6200 series processors Linux tuning guide*, 2012.

[108] Jens Axboe. Flexible I/O Tester: http://freecode.com/projects/fio.

[109] AMD Inc. HyperTransport 3.0 Specification, 2012. Available at http://www.hypertransport.org/default.cfm?page=HyperTransportSpecific-ations3.

[110] Advanced Micro Devices Inc. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 11h Processors*, 2008. Available at http://support.amd.com/us/Processor_TechDocs/41256.pdf.

[111] E. Jeannot, G. Mercier, and F. Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2013.

[112] M.E. Acacio, J. Gonzalez, J.M. Garcia, and J. Duato. A two-level directory architecture for highly scalable cc-NUMA multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 16(1):67–79, 2005.

[113] D. Wang and X. SUN. APC: A novel memory metric and measurement methodology for modern memory system. *Computers, IEEE Transactions on*, PP(99):1–1, 2013.

[114] Selcuk Karabati and Panagiotis Kouvelis. A minsummax resource allocation problem. *IIE Transaction*, 32(3):263–271, 2000.

[115] Roland W. Freund and Florian Jarre. Solving the sum-of-ratios problem by an interior-point method. *J. of Global Optimization*, 19(1):83–102, 2001.

[116] H.M. Monti, A.R. Butt, and S.S. Vazhkudai. Catch: A cloud-based adaptive data transfer service for hpc. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1242–1253, 2011.

[117] B.W. Settlemyer, J.D. Dobson, S.W. Hodson, J.A. Kuehn, S.W. Poole, and T.M. Ruwart. A technique for moving large data sets over high-performance long distance networks. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–6, 2011.

[118] B. Eckart, Xubin He, Qishi Wu, and Changsheng Xie. A dynamic performance-based flow control method for high-speed data transfer. *Parallel and Distributed Systems, IEEE Transactions on*, 21(1):114–125, 2010.

[119] Lustre online page. Available at "http://lustre.org".

[120] Ganglia online page. Available at http://ganglia.sourceforge.net/.

[121] Graphite online page. Available at http://graphite.wikidot.com/.

[122] W. Allcock and J. Bresnahan. Maximizing your globus toolkit gridftp server. *CLUSTERWORLD*, 2:1–7, 2004.