Studies in Exascale Computer Architecture: Interconnect, Resiliency, and Checkpointing

by

Sandunmalee Nilmini Abeyratne

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2017

Doctoral Committee:

Professor Trevor N. Mudge, Co-Chair
Assistant Professor Ronald Dreslinski Jr., Co-Chair
Professor David Blaauw
Professor Chaitali Chakrabarti, Arizona State University
Assistant Professor Reetuparna Das

"You doubtless will make some mistakes, just as we do, and just as everybody else does, but if we all worked on the assumption that what is accepted as true is really true, there would be little hope of advance."

— Orville Wright, *in a letter to George A. Spratt*

Sandunmalee Abeyratne

sabeyrat@umich.edu

ORCID iD: 0000-0002-5716-7595

This dissertation is dedicated to my parents.

Thank you Amma and Thaththa, for believing in me!

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my primary research advisor, Professor Trevor Mudge, for granting me this unique opportunity to do research. He provided me with abundant resources and took a hands-off advising approach that gave me plenty of freedom to delve into my own research interests. Professor Ron Dreslinski, my co-chair, has been immensely helpful in breaking down and understanding the more confounding research ideas. His help in the writing process, creating presentations, and giving talks has been invaluable. I would like to thank my friend and advisor Professor Reetu Das for taking me under her wing during my first two years and patiently teaching me the ropes of research. Late nights spent writing and running simulations on the kilo-core paper and continuously pushing me to create the best conference presentation has not been forgotten. I am fortunate to have had the opportunity to work with Professor David Blaauw, whose sharp mind and insight was always refreshing. Although I met Professor Chaitali Chakrabarti in the last years of graduate school, she helped me immensely in finishing the latter part of this thesis, even making Skype calls half way around the world. Our weekly, and sometimes daily, phone conversations and help in editing multiple revisions of papers, even late at night, were enormously useful.

I would like to thank many of my collaborators for their help in co-authoring papers and developing the research ideas that made this dissertation possible. From the first day Byoungchan Oh joined our lab, his knowledge and mentorship has been crucial to my learning and creating of new ideas. His support has given me much

confidence. I enjoyed our lively debates, late nights spent dissecting papers, and many lessons about Korea. I am grateful for my fruitful collaboration with Hsing-Min Chen, whose expertise filled in the gaps of my own knowledge, and thankful for his help in writing some of the more challenging parts of this dissertation. Supreet Jeloka has been my go-to person for reasoning about the realities of circuit design; I very much valued both his expertise and friendship. I am also thankful to Qingkun Li, Bharan Ghridhar, Sparsh Mittal, Jeffrey S. Vetter, and many other collaborators for their time and advice.

Most importantly, I would like to thank my parents and my grandparents for their unwavering love and support. From the first day, they have instilled in me the importance of higher education and the power of knowledge. My father has always been my greatest cheerleader, never doubting my potential to reach ever-greater successes. His financial support, constant encouragement, and wise advice has opened up many opportunities for my future that otherwise would not have been possible. I owe much of my successes to my mother and her endless encouragement. She is my first teacher and greatest role model, and always helped with my schooling in every way that she can. Her 'never stop learning' attitude towards her own life has been been a huge inspiration. Thank you both for the untold number of sacrifices you have made on my behalf. I know that my grandparents, if they were here today, would be proud and jubilant of my accomplishments and the journey I have made. Their kindness and love will be ever-present in my heart.

I am forever grateful to the emotional support from my TRON lab colleagues Korey Sewell, Tony Gutierrez, Qi Zheng, Johann (Jojo) Hauswald, Yajing Chen, CaoGao, Yiping (Yipee) Kang, Jon Beaumont, Tom Manville, and Mike Cieslak. Over the years, many of you have taken on the roles of advisors, collaborators, brothers, and dearest friends. Through the countless hours spent together and the many ups and downs you have become my second family.

I am pleased to have met many brilliant colleagues in ACAL: Shruti Padmanabha, Andrew Luekfahr, Yatin Manerkar, Aasheesh Kolli, Neha Agarwal, Shaizeen Aga, Akshitha Sriraman, Ankit Sethia, Gaurav Chadha, and Ram Kannan. I am grateful for your academic support and continued friendship. I was very lucky to find a very lively group of friends from the very beginning of my time in Ann Arbor. Thank you to Erik Brinkman, Elaine Wah, Aarthi Balachander, Dev Goyal, Sai Gouravajhala, Lauren Hinkle, Sanae Rosen, Rob Goeddel, James Kirk, Travis Martin, Zach Musgrave, and many others for keeping me sane through it all with many dinners, baking, field trips, house parties, and karaoke nights.

One of my most rewarding experiences of graduate school has been the time I spent with the talented ladies of CS KickStart Meghan Clark, Catherine Finegan-Dollak, Laura Wendlandt, and Katie Hennells. The seed of change that we planted will have enormous impact on the lives of young ladies and the future of CS at Michigan.

When research and graduate school life got tough, my diversions came in the form of many flights into the skies over Ann Arbor. For that, I must thank my flight instructors, Kellen Chong and Joe Burkhead, for their enthusiasm and patience in teaching me the skills and joys of flight and to the pilots at Solo Aviation and Michigan 99s who welcomed me into their community. I also thank all my friends who trusted my skills and let me take them flying.

Finally, my research was made possible by the generous financial support of the CSE department, ARM Ltd, Huawei Technologies, and the DOE Blackcomb Project. Thank you to CSE departmental staff, including Kelly Cormier, Robyn Bollman, Ashley Andreae, Densie DuPrie, Stephen Reger, and many others for their assistance throughout the years. Thank you to everyone who helped me get here!

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF ABBREVIATIONS

CLC Checkpoint Location Controller

DIMM dual in-line memory module

DRAM dynamic random access memory

ECC Error Correction Codes

HPC high performance computing

LAN Local Area Network

MATT Memory Accesses Tracking Table

MPI Message Passing Interface

MTBF mean time between failures

PCIe Peripheral Component Interconnect Express

PFS parallel filesystem

SATA Serial ATA

SSD Solid State Drive

ABSTRACT

Studies in Exascale Computer Architecture: Interconnect, Resiliency, and Checkpointing

by

Sandunmalee Nilmini Abeyratne

Chairs: Trevor N. Mudge and Ronald Dreslinski Jr.

Today's supercomputers are built from the state-of-the-art components to extract as much performance as possible to solve the most computationally intensive problems in the world. Building the next generation of *exascale* supercomputers, however, would require re-architecting many of these components to extract over 50× more performance than the current fastest supercomputer in the United States. To contribute towards this goal, two aspects of the compute node architecture were examined in this thesis: the on-chip interconnect topology and the memory and storage checkpointing platforms.

As a first step, a skeleton exascale system was modeled to meet 1 exaflop of performance along with 100 petabytes of main memory. The model revealed that large kilo-core processors would be necessary to meet the exaflop performance goal; existing topologies, however, would not scale to those levels. To address this new challenge, we investigated and proposed asymmetric high-radix topologies that decoupled local and global communications and used different radix routers for switching network traffic at each level. The proposed topologies scaled more readily to higher numbers

of cores with better latency and energy consumption than before.

The vast number of components that the model revealed would be needed in these exascale systems cautioned towards better fault tolerance mechanisms. To address this challenge, we showed that local checkpoints within the compute node can be saved to a hybrid DRAM and SSD platform in order to write them faster without wearing out the SSD or consuming a lot of energy. A hybrid checkpointing platform allowed more frequent checkpoints to be made without sacrificing performance. Subsequently, we proposed switching to a DIMM-based SSD in order to perform fine-grained I/O operations that would be integral in interleaving checkpointing and computation while still providing persistence guarantees. Two more techniques that consolidate and overlap checkpointing were designed to better hide the checkpointing latency to the SSD.

CHAPTER I

Introduction

Supercomputers work on the most compute intensive applications in the world that require rigorous mathematical calculations and data processing. In the United States and abroad, supercomputers are used by governments and research institutions to solve a vast range of scientific problems. For instance, they are used to study weather patterns and predict storms, discover oil and gas, study atoms and particles, and most recently to develop precision medicine. Institutions around the world compete to build powerful supercomputers and they are ranked by the Top 500 List, which ranks twice annually the 500 fastest supercomputers in the world by their peak floating point operations per second (FLOPS) rate [9]. The international race to build the fastest supercomputer is not just a matter of national pride, but also pivotal to the technological advancement of each country. At the time of this writing, two of China's custom designed supercomputers, Sunway TaihuLight and Tianhe-2 (MilkyWay-2), take the top 2 spots, followed by three supercomputers belonging to the U.S. Department of Energy, Titan (Cray XK7), Sequoia (IBM BG/Q), and Cori (Cray XC40).

Performance is by far the most important goal in building these systems, followed by keeping costs and power consumption to a minimum. For the next decade, the biggest milestone for the supercomputing community is to build an *exascale* super-

computer with the ability to compute one exaFLOP[1] (10^{18} floating point operations) per second. When built, this supercomputer will be 50× faster than Titan, currently the fastest supercomputer in the United States whose peak performance is roughly 20 petaflops [21]. This high performing supercomputer will be built from linking together thousands, maybe hundreds of thousands, of high performing *compute nodes*.

To better understand the scale of the future supercomputer, a skeleton exascale system that would meet 1 exaflop of performance and have 100 petabytes of dynamic random access memory (DRAM) main memory was modeled. This skeleton design, shown in Figure 1.1, was inspired by DARPA's *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems* [19]. Some of the main parameters were kept similar in value with DARPA's design such as the 6 Gflops of performance per core tile, the number of cores per node, and the total number of nodes. This is just one design point among many possible exascale designs.

As outlined in the figure, we estimated 768 cores per compute node, which far exceeds multi-core and many-core processors with 10 to 100 cores [4, 5, 7, 8, 10] in the market today. Although processors with fewer cores can be used, that approach would have to be compensated with adding more compute nodes in order to attain the same exaflop performance goal. More compute nodes occupy more real estate (cabinets, racks, floor space), need more Local Area Network (LAN) switches, and introduce more wiring.

Even after incorporating bigger kilo-core processors, future exascale supercomputers will still have millions of other components. For instance, according to the skeleton model, the entire system would have 204,800 compute nodes. This number far exceeds the 18,688 nodes in Titan. Furthermore, at least 100 petabytes of DRAM-based main memory would be required to support the amount of data being processed by all the cores. Even when using the industry's highest capacity 128GB DDR4 modules,

[1] also exaflop

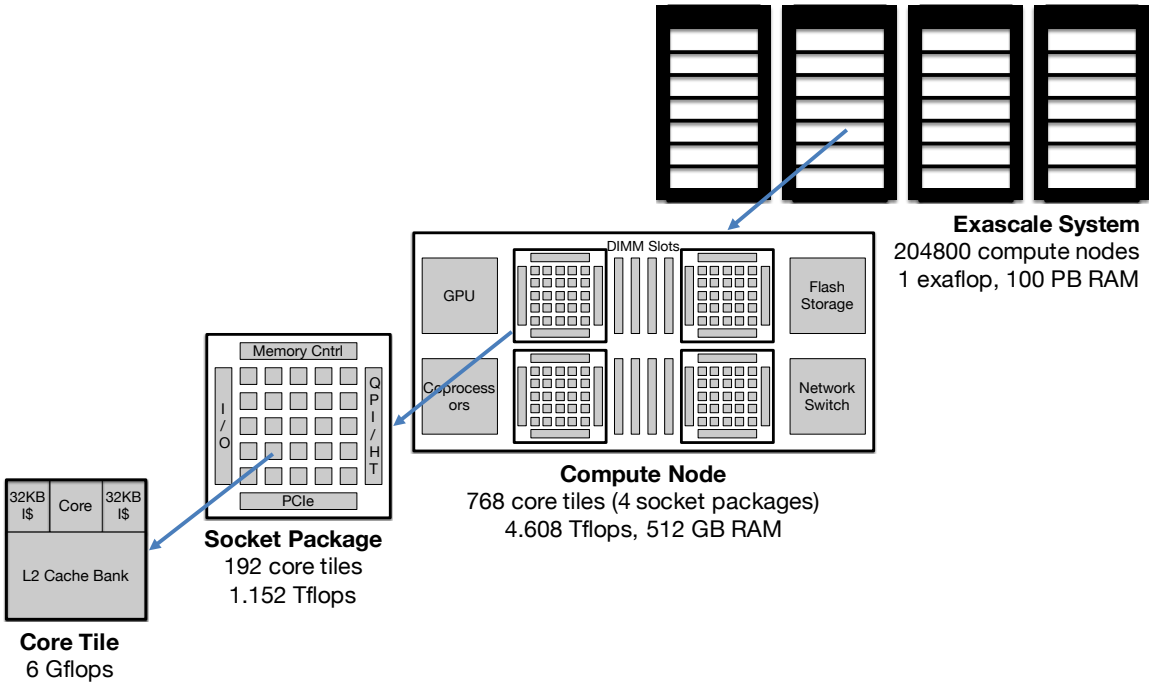


Figure 1.1: A skeleton exascale architecture

819,200 modules are required. Additional switches, routers, and power rails are also needed to support the increase in number of nodes.

1.1 Problem Space

In this dissertation, we studied two problems regarding the architecture of an exascale compute node. The first research problem targeted on-chip interconnect topologies for kilo-core processors. The layout of a processor with a thousand cores looks vastly different than one with just 10 cores. For instance, wire length from cores at the center of the chip to the edge peripherals can be 10× longer than wire lengths from edge cores. In this study, we asked the question whether existing topologies used for many-core processors will scale for kilo-core processors. After making the observation that kilo-core processors benefit from decoupling local communication to nearby cores and global communication to faraway cores, we proposed two highly scalable on-chip interconnect topologies called *Super-Star* and *Super-StarX* for kilo-

core processors. These topologies were designed to improve interconnect throughput, reduce packet latency, and reduce interconnect energy consumption.

The second research problem we studied was checkpointing for fault tolerance. Due to the vast number of components present in the system, it is inevitable that there will be many failing components that must be handled gracefully with quick recognition and recovery. Fault tolerance is imperative to supercomputers because scientific applications have lengthy computations that can take days, weeks, or even months to complete. Checkpointing the application's progress periodically to a stable, non-volatile storage device aids in maintaining its progress even in the midst of failures such as power outages, cosmic rays, software bugs, etc. Checkpointing has other uses in supercomputing as well such as record-replay debugging and post-processing visualizations. In this study, we asked the question whether a local Solid State Drive (SSD) can be used as a checkpoint storage platform without degrading performance and wearing out. After making the observation that DRAM main memory can help offset costs of checkpointing to the SSD, we proposed a hybrid DRAM and SSD solution for local checkpointing at the compute node. This hybrid solution was designed to improve the speed of checkpointing and to reduce its energy consumption by trading off some tolerance against power failures.

The final part of my dissertation centered on hiding the checkpointing latency to the SSD. Current ways of writing I/O data to an SSD involves the operating system and incurs huge slowdowns to guarantee the data has been fully persisted. After making the observation that dual in-line memory module (DIMM)-based SSDs offer a tighter coupling between main memory and storage as compared to conventional Serial ATA (SATA) or Peripheral Component Interconnect Express (PCIe)-attached SSDs, we proposed a fine-grained data copying method between main memory and SSD storage that tracks and copies only the modified parts of the data. Additionally, we propose two techniques called *consolidation* and *early-late overlapping* to more ef-

fectively hide the enormous performance degradation incurred by conventional checkpointing methods. These optimizations were designed to further improve the speed of checkpointing while reducing main memory bandwidth and main memory space consumption by checkpoints.

The next three sections will introduce the three main works of this dissertation.

1.2 Asymmetric High-Radix Topologies

In the first work, we explored the challenges in scaling existing on-chip network topologies towards kilo-core processors. Current low-radix topologies such as mesh optimize for fast local communication, but do not scale well to kilo-core processors because of the large number of routers required. These increase both power and hop count. In contrast, symmetric high-radix topologies such as concentrated mesh and flattened butterfly optimize for global communication with fewer hop counts, but degrade local communication with their large, slow routers.

To address both local and global communication optimizations independently, we decoupled the interconnect design by using asymmetric high-radix topologies. By setting a design goal of matching router speed with wire speed, our proposed topologies use fast medium-radix routers to optimize for local communication and a few slow high-radix routers that reduce hop count to optimize for global communication. Our asymmetric high-radix designs are enabled by recently proposed *Swizzle-Switches*, which allow us to achieve performance scalability within realistic power budgets.

We proposed two asymmetric high-radix topologies: *Super-Star* (asymmetric folded Clos) and *Super-StarX* (asymmetric folded Clos with superimposed mesh). The new topologies were evaluated on a chip with 552 cores and 24 memory controllers. The cores were modeled after an out-of-order ARM Cortex A15 core and laid out on a 24×24 grid. Our evaluations show that the best performing asymmetric high-radix topology improves average network latency over a mesh topology by 45% while re-

ducing the power consumption by 40%. When compared to symmetric high-radix topologies, network throughput improves by 2.9× while still providing similar latency benefits and power efficiency.

This first work was published in the Proceedings of the International Symposium on High Performance Computer Architecture (HPCA) in 2013 [11].

1.3 Hybrid Checkpointing to DRAM and SSD

In the second work, we explored the challenges of checkpointing fast and reliably to the compute node's storage. Checkpoint/restart is a key ingredient in attaining resilience, but it is becoming more challenging as the amount of data to checkpoint and the number of components that can fail increases in exascale systems. To improve the speed of checkpointing, emerging non-volatile memory (phase change, magnetic, resistive RAM) have been proposed. However, using unproven memories to create checkpoints will only increase the design risk for an exascale memory system. In this work, we showed that exascale systems with hundreds of petabytes of memory can be constructed with commodity DRAM and SSD flash memory and that newer non-volatile memory are unnecessary, at least for the next generation.

The challenge when using commodity parts is providing fast and reliable checkpointing to protect against system failures. A straightforward solution of checkpointing to local flash-based SSD devices will not work because they are endurance and performance limited. Hence, we presented a checkpointing solution that employs a combination of DRAM and SSD devices. A Checkpoint Location Controller (CLC) is implemented to monitor the endurance of the SSD and the performance loss of the application and to decide dynamically whether to checkpoint to the DRAM or the SSD.

The CLC improves both SSD endurance and application slowdown; but the checkpoints in DRAM are more exposed to failures because Error Correction Codes (ECC)

in DRAM are weaker than those in SSDs. In order to protect data in DRAM, we proposed a low latency ECC that can correct all errors due to bit/pin/column/word faults and also detect errors due to chip failures, and we protected the checkpoint with a Chipkill-Correct level ECC that allows reliable checkpointing to the DRAM. The two ECC mechanisms were developed jointly with researchers at Arizona State University [12].

Using our system, the SSD lifetime increases by 2×—from 3 years to 6.3 years. Furthermore, the CLC reduces the average checkpointing overhead by nearly 10× (47% from a 420% slowdown), compared to when the application always checkpointed to the SSD.

This second work was published in the Proceedings of the Second International Symposium on Memory Systems (MEMSYS) in 2016 [12].

1.4 Improvements to Checkpointing with a DIMM-based SSD

In the third work, we explored the challenges of hiding the checkpointing latency to the SSD. Writing a checkpoint to the SSD requires a guarantee in return that it has been persisted to the device in case a power failure later on wipes out the data. Currently, applications that require persistence must employ an `fysnc()` operation and expose the copying latency of the entire data set from main memory to storage. Conversely, employing a background thread or process to hide I/O latency triggers copy-on-write semantics that not only uses additional memory space to create duplicate copies but also uses additional memory bandwidth for in-memory copying.

In order to address this problem, we proposed to directly write-protect memory pages with checkpoint data and incrementally copy that data from main memory to storage in parallel with ongoing computation. In order to engineer the incremental copying of data from memory to storage, we use newer DIMM-based SSDs. The memory-bus attached DIMM-based SSD is able to track modified cache lines of mem-

ory at a fine granularity via the shared memory controller. During I/O operations, the SSD controller is able to request only modified cache lines for saving, thereby reducing I/O latency and traffic.

Using the proposed fine-grained copying method, we additionally propose to *condense and consolidate* multiple modified main memory pages into a checkpoint with a few flash pages. Consolidating amortizes the slow programming latency of a flash page over as much checkpoint data as possible. Applying consolidation on top of the conventional stop-and-copy method speeds up checkpointing by 41% on average. We proposed a second method called *early-late* checkpointing to overlap data copying with computation by starting checkpointing of select memory pages earlier than the beginning of the checkpoint phase and continuing to checkpoint them well into later computation phases. Applying the early-late method speeds up checkpointing by 32% on average over stop-and-copy. Together, the proposed consolidating and overlapping methods provides a 79% speedup.

A faster checkpoint latency to the SSD implies that more checkpoints can be written there (endurance allowing) such that less progress will be lost in a failure. Consolidated checkpoints also reduce checkpointing energy and space.

1.5 Dissertation Organization

The research work is presented in five self-contained chapters. Chapter II gives background information of fault tolerance, checkpoint/restart, non-volatile memories and flash. Chapter III presents the study on asymmetric on-chip network topologies. Chapter IV presents the work on hybrid checkpointing to commodity DRAM and SSD platforms. Chapter V discusses further improvements to hiding checkpoint latency with a DIMM-based SSD. Finally, Chapter VI includes the summary of this work and conclusions.

CHAPTER II

Background

This chapter provides some background into fault tolerance, checkpoint/restart, and flash memory.

2.1 Fault Tolerance in High Performance Computing

We follow the terminology set by Sridharan et. al in distinguishing between errors and faults [107, 108, 109]. A **fault** is the underlying cause of an error such as a dead component and an **error** is a symptom or a manifestation of the fault such as an incorrect value produced by the dead component. Fault tolerance is imperative to ensure that an application successfully finishes with the correct result. If the application ended with the wrong output or crashed, then that is an obvious sign of a fault somewhere in the system. The application could, however, produce the correct result after prolonged execution indicating that there might be a silent data corruption (SDC) [16] fault present in the system. Applications such as *miniFE* that iterate until the value converges within an error tolerance exhibit this behavior.

Faults can appear in many places such as the program code, any piece of software at any layer, and hardware components (cores, registers, caches, memory, storage, network cards, network cables, motherboard, power supply, etc). Errors can be caused by software bugs, hardware bugs, environmental factors (e.g. power outages, cos-

mic rays), human errors, and by many other unknown origins [99]. Achieving fault tolerance has to be a combination of detection, identification, correction, and prevention. There are several works in literature by author Bianca Schroeder that study the failures seen in high performance computers [97, 98, 99]. Ideally, each of these hardware components should have built-in many ways to detect, identify, and correct errors. But this is not a straightforward solution due to the overhead of additional circuitry, performance slowdown, and higher cost. Furthermore, sometimes errors are correctable—as commonly seen in memory structures that use error correction codes—but other times they necessitate replacing the failed part.

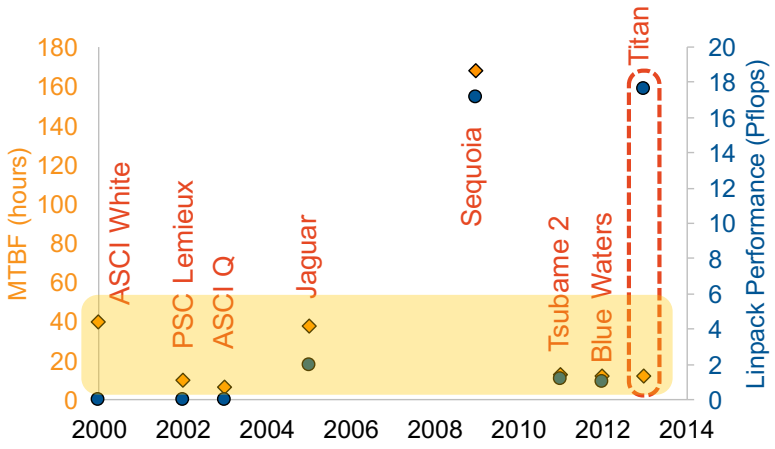


Figure 2.1: Timing overheads by conventional checkpointing methods

Figure 2.1 plots the mean time between failures (MTBF) value and the peak Linpack performance of some supercomputers deployed in the United States over the past decade and a half. It can be seen that over the years the MTBF has remained at or below 40 hours (with the exception of Sequoia). This indicates that even if the fault tolerance ability of individual components has gotten better, the increasing number of components in a system leads to the same aggregate failure rate.

Achieving resilience is becoming an even bigger problem at exascale due to the following reasons [38]:

- Number and variety of components are increasing. Heterogeneous architec-

tures such as CPUs, GPUs, many-core co-processors, accelerators and such are required if we are to ever achieve the exascale goal. They introduce different types of errors and different failure rates. Adding error detection and correction circuitry to each component becomes expensive.

- Transistors are becoming smaller. Smaller device sizes and lower voltages increase vulnerability to soft errors. Transistors that are only a few atoms thick are easily upset by cosmic rays. Smaller feature sizes also inherently have larger process variation, which results in occasional inconsistent behavior. Smaller transistors and wires also age more rapidly and unevenly leading to frequent permanent failures.

- Adding resilience is expensive. Strengthening components against failures by adding hardware resilience requires adding more circuitry, making the components more expensive. In addition to cost, the power may also be higher.

2.2 Checkpoint/Restart

The most common approach to fault tolerance in high performance computing (HPC) is checkpoint/restart. It is one of the most intuitive and simplest ways of surmounting failures of any type. The idea is to simply restart the program after a failure. However, rather than restarting from the beginning, the idea is to restart from the last known point of correct execution. Often, restart is assumed to take place after the failed component has been corrected or replaced.

2.2.1 Problems with Checkpointing

The most troublesome portion of checkpoint/restart is creating the checkpoints themselves. Since failures are often unexpected, both in the time of their occurrence and in their severity, it is desirable to checkpoint as often as possible to be well-

prepared to recover to the most recent correct state of execution. The goal is to re-do as little work as possible in order to minimize execution time. However, the cost of checkpointing is that it has to be mutually exclusive from application execution. In other words, in order to capture a consistent application state (i.e. a snapshot of applications state frozen at a moment in time), the program cannot be both executing and checkpointing at the same time. Since checkpointing while updating data could lead to capturing an unstable state of execution.

Since the checkpointing time is non-trivial, constantly stopping the program to create checkpoints add noticeable slowdown to any HPC application. In the future, as supercomputers run bigger applications with more data across more compute nodes, the slowdown due to checkpointing will only worsen. Therefore, there is an inevitable tradeoff between the amount of time spent on checkpointing vs. the progress lost as a result of rolling back to a distant past.

2.2.2 Efforts to Reduce Checkpointing Overhead

One of the biggest issues of checkpointing is the time overhead associated with it. Ideally, checkpoints should be saved to the most visible and accessible location, which is often the parallel filesystem (PFS). The PFS is hosted on 'storage nodes' which are accessed via 'I/O nodes'. As a side note, the storage nodes, I/O nodes, or the interconnect connecting all the nodes may themselves fail. Writing a system-wide checkpoint originating from multiple Message Passing Interface (MPI) tasks running across hundreds of compute nodes over narrow I/O channels to the PFS adds significant overhead. It has been reported that applications spend as much as 15-30 minutes [27] waiting for a checkpoint to finish. There has been a wealth of research addressing the time overhead. We classify these approaches into several main themes below.

- Reducing distance to checkpoint storage. Rather than sending a checkpoint all

the way to the filesystem, hierarchical checkpointing has been proposed to store the checkpoint at multiple levels of storage, such as at the local compute node, intermediate I/O nodes, and the PFS [18, 33, 74, 92]. Diskless checkpointing proposed to save checkpoints to the storage of other compute nodes [43]. Burst buffers proposed to place storage that can 'absorb the burst of checkpointing traffic' at the I/O nodes [67]. One problem with local checkpointing is that some supercomputers today do not have storage in the compute nodes, only DRAM memory. However, this trend is set to change with Theta (2016), Summit (2018) and Aurora (2019) supercomputers that have announced to place an SSD in each compute node [13].

- Reducing the size of checkpoint. A few ways of reducing the checkpoint size is incremental checkpointing, compression [50, 75], compiler analysis to remove dead variables [25, 26, 83], and application-level checkpointing in which the programmer annotates critical data. All these ideas aim to save only the minimal possible state to ensure correct restart. Reducing the checkpoint size not only reduces time overhead, but also the area/storage overhead necessary to store it.

- Reducing stalled time due to checkpoint. Interleaving or overlapping checkpointing with application execution, lazy checkpointing, and skipping checkpoints are all ways of reducing stalled time. Pages whose data values will not change before the next checkpoint can start checkpointing early, interleaving with computation. This approach reduces the amount of data left to be checkpointed during the actual checkpointing phase. Another, lazy way is to simply mark the checkpoint data with a write-protect flag and have a background thread save them while the application continues execution. Data can be write-protected in hardware by appropriately modifying cache, memory, and/or paging hardware. The `fork()` operation in the Linux kernel can create a back-

ground child process with an identical memory image and its *copy-on-write* semantic automatically duplicates write-protected pages before modifying them. The challenge with this approach is finishing checkpointing as fast as possible so as not to create too many page copies. In the worst case, *copy-on-write* could double the memory footprint of the checkpointing application. Once the checkpoint is finished, the child process can be killed. Finally, the compiler or the runtime system can decide to skip some checkpoints that are unlikely to be needed for recovery.

- Use of non-volatile memories. Non-volatile memories are an attractive option for checkpointing for two reasons: 1) they are faster than conventional hard disk drive storage and 2) they are non-volatile. Checkpoints are always stored on a non-volatile platform to ensure persistence across reboots. Non-volatile memories such as flash, STT-RAM, PCM, and resistive memories are orders of magnitude faster than hard disk drives, and in some cases like STT-RAM, they are as fast as DRAM.

2.2.3 Non-volatile Memories for Checkpointing

There are many types of non-volatile memories, but the most commonly heard of are NAND and NOR flash, ferroelectric (FeRAM or FRAM), magnetic (MRAM), phase-change (PCM or PCRAM), and resistive (ReRAM or RRAM). Within this list of NVMs, there are two distinct groups by the maturity of the technology. Flash is the most mature technology while the rest are called "emerging non-volatile memories." Flash exists in many forms in the market already, some common products are SSDs, SD memory cards for cameras and mobile phones, and USB thumb drives. Flash products have been widely popular for over a decade and its properties, behavior, and failures are well-understood. Furthermore, its manufacturing process is well-established, making flash products incredibly inexpensive compared to the emerging

NVM technologies, although they are still more expensive than hard disk drives.

The "emerging" group of technologies are still being developed by research and industry alike. The two biggest advantages of emerging technologies are that 1) they are all faster than flash and 2) they all have more write endurance/lifetime than flash. There are no commercial products on the market for them, but companies such as Intel, Micron, SanDisk, Toshiba, SK Hynix, and HP Labs have either attempted or are currently attempting to make them into usable products. HP Labs started building memristors as far back as 2008. In 2015, one year after ambitiously announcing their plan to build a memristor-based machine, named the 'Machine', HP Labs decided to remove the memoristors from their Machines until further notice because it was not economically viable for volume production. A small Santa Fe startup beat HP to the market by putting the first memoristor chip on the market priced at $220 each for experimental uses.

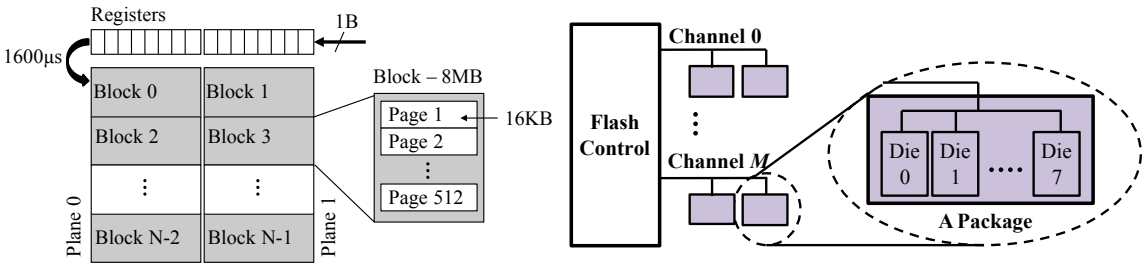
While memristors are making slow progress, a notable program that has been garnering a lot of media attention lately is 3D XPoint from Intel. Despite speculation from industry experts, Intel has yet to reveal the real technology behind 3D XPoint. Intel's Optane SSD based on 3D XPoint technology was released at the beginning of 2017. Undoubtedly, the biggest issues plaguing non-volatile memories are: 1) it is unclear when we can produce them in commercial quantities, 2) it is unclear how much they will cost, and 3) it is unclear what problems they will face over the long-term in the hands of consumers.

2.2.3.1 Flash Memory Operation

In this dissertation, we advocate flash-based SSDs for checkpointing. Commercial availability and maturity of NAND-flash prove them a low-risk option sufficient for at least the first generation of exascale systems, if used correctly. Therefore, in this section we provide some background into how flash memory operates.

The internal organization of a flash die is shown in Figure 2.2a. 512 16KB pages are organized into blocks of 8MB, several of which are grouped into planes. A single flash die contains one or two planes. The smallest programmable unit is a single page and it takes 1600μs for a 16KB page [71]. Two pages (a plane pair) can be programmed at once if there are two planes in the die and a bandwidth of 19.5 MB/s per NAND flash die can be achieved.

SSDs exploit three dimensions of parallelism to improve the bandwidth to flash devices (Figure 2.2b). First they use *plane-level parallelism* in which 1 to 4 planes are operated on simultaneously for read/write/erase operations. Second, they use *die-level parallelism* in which each package contains as many as eight dies and each die can be operated on independently. Thirdly, they use *channel-level parallelism* in which multiple packages are connected to the SSD controller over different channels and these channels can be operated on individually and simultaneously.



(a) Die

(b) Controller, channels, packages, and dies

Figure 2.2: Flash organization

Table 2.1 shows a sampling of raw NAND flash devices sold by manufacturers Micron, Toshiba, and Samsung. It shows that typical flash program latencies range from 220μs to 1.6ms for page sizes ranging from 2KB to 16KB. The write bandwidth of a single flash device is less than 25MB/s at best. In contrast, commercially available SSDs market up to 450MB/s to 2GB/s for SATA and PCIe connections, respectively. Comparatively, DRAM main memory bandwidths of 12.8GB/s DDR3-1600 and 19.2GB/s DDR4-2400 surpasses even the best SSDs.

Table 2.1: A sampling of commercially available raw flash devices and their reported read, program, and erase latencies.

Manufacturer & Year	Device Size (Gb)	Page Size & Spare Area	Read (μs)	Program (μs)	Erase (μs)
SLC Types [72, 112]					
Micron 2006	4/8/16	2KB + 64B	25	220	1500
Micron 2010	4	4KB + 224B	25	200	2000
Toshiba 2013	16	4KB + 232B	30	300	3000
MLC Types [71, 72]					
Micron 2005	16/32/64/128	4KB + 218B	50	900	3500
Micron 2009	64/128/256/512	8KB + 448B	75	1300	3800
Micron 2013	128/256/512/1024	16KB + 1216B	115	1600	3000
3D V-NAND [55, 90]					
Samsung 2014 2-bit	128Gb (24 layers)	8KB + 698B	49	600	4000
Samsung 2014 2-bit	128Gb (32 layers)	16KB + 1536B	35	390	4000
Samsung 2015 3-bit	28Gb (32 layers)	16KB + 1536B	45	700	3500
Samsung 2016 3-bit	256Gb (48 layers)	16KB + 1536B	45	660	3500

CHAPTER III

Asymmetric High-Radix Topologies

3.1 Introduction

Today's chip designers have resorted to increasing the number of cores in a chip as a power-efficient approach to throughput scaling. Processors with 10 to 100 cores [4, 5, 7, 8, 10] are already in the market today, and a processor with 1000 cores (kilo-core) may soon be a reality. While off-chip interconnection networks for 100s of nodes have been studied in the past, a power and performance scalable on-chip network for a kilo-core chip is a new challenge.

If we use a conventional topology constructed out of low radix routers[1], such as a 2D-Mesh [7, 48, 91, 111, 119], then the number of routers required increases as the number of cores increases. The power consumption of this growing number of routers coupled with the decreased performance resulting from larger hop counts will soon become prohibitive.

One solution to this problem is to consolidate routers into a few large but efficient high-radix switches. While high-radix switch designs were thought to be impractical due to the power and area complexity, recent work with the Swizzle Switch design [36, 94, 95, 96, 103] has demonstrated that on-chip high-radix switches are feasible. The *Swizzle-Switch* is shown to scale up to a radix of 64 while supporting

[1]Radix is defined as the number of ports in a router

128-bit channels, consuming less than 2W of power and operating at a frequency of 1.5GHz in 32nm technology. High-radix topologies facilitated by *Swizzle-Switches* make it possible to design scalable on-chip networks for kilo-core processors within realistic power budgets. A high-radix switch can be utilized to improve scalability of interconnects in kilo-core chips by concentration [17], where multiple cores/nodes share a router, thereby reducing the number of routers and network diameter. Also, high-radix switches can be used for designing a topology which provides more physical express links between non-adjacent routers [59], again reducing the network diameter. However, there are two problems with these approaches. First, using concentration to scale common designs (e.g., 2D-meshes), leads to lower network throughput because of bandwidth bottlenecks in inter-router links. Second, spatially close-by nodes are communicating through slower high-radix switches, degrading the performance of local communication. Thus, conventional high-radix topologies trade-off performance of local communication between close-by nodes for improving performance of global communication by reducing hop-count between nodes that are farther apart.

Our solution to mitigate these problems is an asymmetric high-radix topology. The key design principle of such topologies is to match the frequency of the routers with the length of the wires that connect them. For local communication, wires are short and hence wire delay is small. Therefore, the routers that facilitate local communication should operate at a higher frequency and lower radix to ensure that both wire and router delays are balanced and neither dominates overall latency. Since communication is local, low hop count is maintained even with lower radix routers. In contrast, global communication inherently spans long distances and hence incurs large wire delay. The global router can afford to be slow because the wire latency will be large at most frequencies. Thus, the router frequency can be reduced and its radix can be increased. To offset the effect of the slower router, the high radix of the global router ensures that the number of hops is reduced, which is important for

lowering network latency for global communication.

Based on the above design principle, we propose two asymmetric high-radix topologies for kilo-core processors: *Super-Star* and *Super-StarX*. *Super-Star* is a hierarchical star topology in which a cluster of nodes are connected to a fast medium-radix local router. All local routers are connected by a high-radix global router. The network diameter is two hops. To increase network throughput we duplicate the global routers and there is no connection between the global routers. *Super-Star* with multiple global routers has the same connectivity as a folded-Clos topology [57] with one middle stage. Unlike current on-chip implementations of folded-Clos which assume equal radix routers, we explore *Super-Star* with high-radix global routers and low-radix local routers.

The second design, *Super-StarX*, extends the *Super-Star* design to permit adjacent local routers to directly communicate with each other instead of going through a global router, which further improves the performance of local communication. This optimization increases the radix of local routers by only four, which does not significantly decrease the frequency of local routers. The connections to global routers remain the same as in *Super-Star* and hence, global communication is as efficient as *Super-Star* with a network diameter of only two hops.

As a comparison point a third design, *Super-Ring*, is a hierarchical ring topology that does not follow our design principle of matching router delay with wire delay. In *Super-Ring*, a cluster of local routers is connected to a medium-radix global router. The global routers are then connected in a ring. The *Super-Ring* provides greater connectivity between global routers compared to *Super-Star* and *Super-StarX*. We show that *Super-Star* and *Super-StarX* topologies, unlike meshes and symmetric high-radix topologies, are energy proportional. Their achieved throughput is proportional to the power consumed. The network throughput and power consumption can be turned up or down by varying the number of global routers. Thus network architects can choose

fewer global routers at design time or power-gate the global routers at run-time. It is possible to power-gate the global routers because even a single global router assures full network connectivity. We model a processor with 576 nodes in 15nm technology. This model provides a reasonably large system to study the scalability of interconnect topologies towards future kilo-core chips. We study the proposed network designs through detailed floor-planning, circuit-level delay analysis of routers and wires, network power models, and micro-architectural cycle accurate performance simulations. We study statistical traffic, and also 44 different benchmarks with multiprogrammed workloads of single threaded and multi-threaded shared-memory applications.

Our evaluations show that the best performing asymmetric high-radix topology improves average network latency over a mesh by 45% while reducing the power consumption by 40%. When compared to symmetric high-radix topologies (i.e. concentrated meshes and flattened butterfly), our proposed topologies improve network throughput by 2.9× while still providing similar latency benefits and power efficiency. Over a varied set of application workloads, the final proposed topology improves application performance by 17%, while reducing power consumption by 39%.

In summary, our key contributions are:

- We propose asymmetric high-radix topologies for performance and power scalable on-chip networks for designing kilo-core systems. Our proposed topologies optimize for both local and global communication.

- Our key design principle for asymmetric topologies is to match router speed with wire speed. Fast medium-radix routers support local communication along short wires and a few slow high-radix routers support global communication by reducing hop count. The global high-radix routers can afford to be slow because wire delays of global routes are inherently longer.

- Based on our design principle, we propose and evaluate two asymmetric high-

radix topologies: Super-Star (asymmetric folded-Clos) and Super-StarX (asymmetric folded-Clos with superimposed mesh). These topologies vary in their degree of local and global connectivity.

- We also find that *Super-Star* and *Super-StarX* topologies, unlike meshes and symmetric high-radix topologies, are *energy proportional*.

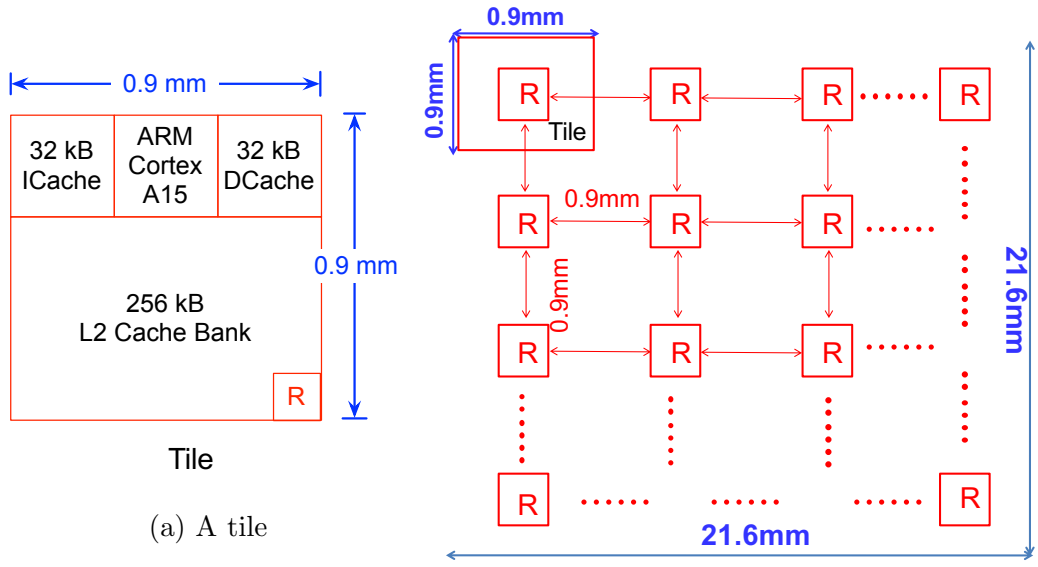
3.2 Motivation and Background

3.2.1 Scaling of Low-Radix Mesh Topology

Low-radix mesh [7, 48, 91, 111, 119] topologies have become popular for tiled manycore processors because of their low complexity, and planar 2D-layout properties. Figure 3.1 shows the layout of a mesh topology. For our studies, we investigate a 576-node chip with 552 core tiles and 24 memory controller tiles. The length of a tile is $0.9mm$. The tile dimensions are chosen such that it can accommodate a simple out-of-order ARM Cortex A15 core, 32 KB of L1 cache, 256 KB of L2 cache and a small radix-5 mesh router in 15nm design. The tiles are connected with a 24×24 2D-mesh.

Unfortunately, as we scale up the mesh topology towards kilo-core processors, it shows poor performance scalability due to its quickly growing network diameter. The large number of routers required by the mesh topology pushes the overall network power far beyond practical limits [23, 24]. High average hop count also leads to high variability of available per core bandwidth [65] and exacerbates worst case latency.

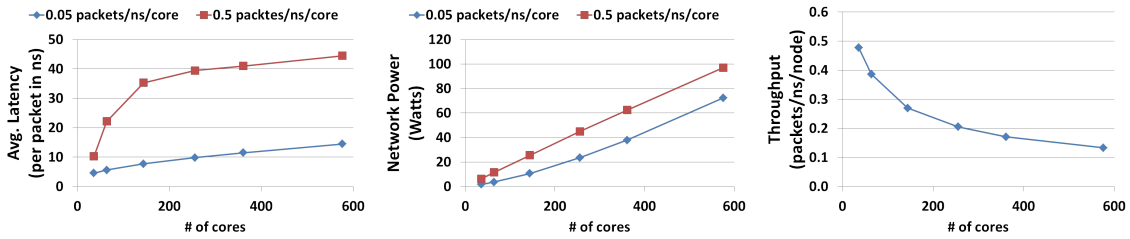
Figure 3.2 illustrates the scaling characteristics of the mesh topology as we increase the number of cores from 36 to 576 (Section 3.4 provides simulation and modeling details). The network latency and power is shown for two injection rates, $0.05 packets/ns/core$ (low) and $0.5 packets/ns/core$ (high). Even at a low injection rate the network latency degrades by 3× as we increase the number of cores from



(a) A tile

(b) Mesh Topology

Figure 3.1: A diagram of a core tile and a mesh topology.



(a) Network latency (b) Network power (c) Throughput per core

Figure 3.2: Scaling of mesh topology with the number of cores.

36 to 576. At the high injection rate, the degradation in latency is steeper. Thus, higher performance afforded by increasing number of cores can be offset by communication overheads. Figure 3.2b shows the steep increase in network power from 6.3W to 97.1W as we increase the number of cores from 36 to 576. Figure 3.2c illustrates that the available per core throughput reduces by 3.7× as we increase the number of cores from 36 to 576. Ideally, we would like the network to provide a constant per-core bandwidth with increasing number of cores, such that the performance of individual cores is not effected by scaling up the number of cores.

The above studies motivate the need for a scalable interconnect topology. It can be seen that future manycore processors cannot afford the luxury of a low-

complexity mesh topology. In this work, we propose asymmetric high-radix topologies as a solution. Before we delve into high-radix topologies, we give a brief background on the *Swizzle-Switch*, which is the key-enabler of our designs. For more details on implementation of the *Swizzle-Switch*, we refer the reader to recent prior work [36, 94, 95, 96, 103].

3.2.2 Enabling High-Radix Routers with *Swizzle-Switch*

The SRAM-inspired design of the *Swizzle-Switch* provides good scalability to large radices. Traditional matrix-style switches consist of a crossbar that routes data and a separate arbiter that configures the crossbar. This decoupled approach poses two hurdles to scalability: (1) the routing to and from the arbiter becomes more challenging as the radix increases and (2) the arbitration logic grows more complex as the radix increases. Arbiters that need to distribute their arbitration over multiple stages incur the overhead of flip-flops to store the control flow signals. The work done by Passas [80] illustrates the difficulty of implementing a multistage arbiter for a high-radix switch. In Passas work, a radix-128 switch is shown to have a crossbar arbiter that consumes 60% of the total crossbar area and requires three stages to do arbitration.

To overcome these limitations, the *Swizzle-Switch* combines the routing-dominated crossbar and logic-dominated arbiter by embedding the arbitration logic within the switch crosspoints. The *Swizzle-Switch* design reuses input/output buses for arbitration, producing a compact design. The arbitration is done in a single cycle by comparing priority bits that are embedded in the switching fabric. At the end of each arbitration stage, the priority bits are automatically updated by setting and re-setting appropriate priority bits to achieve least recently granted order of arbitration. To reduce power, the Swizzle-Switch uses SRAM-like technology with low-swing output wires and a single-ended thyristor-based sense amplifier. We studied the scalability

of the *Swizzle-Switch* across a wide range of radices. Figure 3.3 shows the frequency and energy per bit transferred of the *Swizzle-Switch* as function of its radix. Even when the radix is increased to 64, the *Swizzle-Switch* with 128-bit channels can continue to operate at a high frequency of 1.5GHz while consuming less than 2W of power. In 32nm technology, this *Swizzle-Switch* requires ~2mm^2 of area.

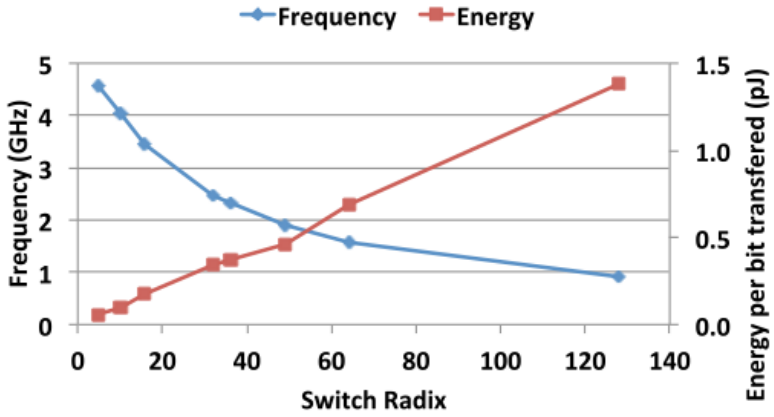


Figure 3.3: Scaling of a 128-bit Swizzle-Switch with radix.

3.3 High-Radix Topology Design

In this section, we explore several high-radix topologies and analyze their scalability in the context of kilo-core processors. First, we discuss symmetric high-radix topologies consisting of all equal-radix routers and their design trade-offs. Then, we discuss asymmetric high-radix topologies where router radix is guided by wire delay. These topologies are designed to optimize both local and global communication

3.3.1 Symmetric High-Radix Designs

3.3.1.1 Concentration

Balfour and Dally [17] proposed a concentrated mesh which allows a few nodes to share a router. The number of nodes sharing a router is called the concentration degree of the router. Since the router is shared, the radix of its switch increases by

least its concentration degree. Concentration yields two benefits: 1) it reduces the network latency by reducing the network diameter and average hop count; and 2) it reduces the number of routers, which can lead to power savings.

However, the benefits of concentration are largely dependent on the power-frequency scalability of the switch. As we increase the concentration degree (and hence the switch radix), the routers become larger and slower in terms of frequency, and the wires which connect them become longer. Thus the benefits due to reduced hop count may be offset due to reduction in performance of individual switches.

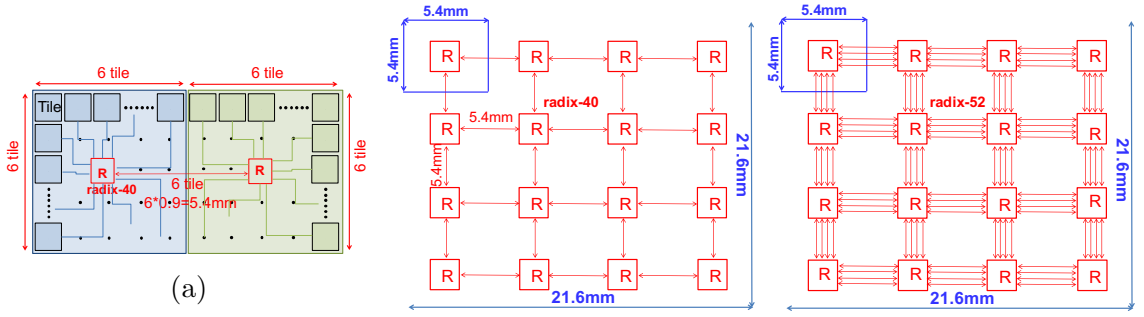


Figure 3.4: Concentrated Mesh Topology: (a) Layout of tiles in a cluster for a concentration degree of 36. (b) Layout of concentrated routers in a mesh. (c) Layout of concentrated mesh with 4 parallel links between routers.

In [17], the authors target a 64-tile system where 4 tiles share a router. We find that a concentration degree of 4 does not provide sufficient scalability for kilo-core systems. To scale to 576 nodes, we leverage Swizzle-Switches to increase the concentration to much higher degrees, and study the trade-offs between reduced hop count and reduced router frequency. Figure 3.4b shows the layout of concentrated mesh with a concentration degree of 36 for our target processor design. Each router services 36 tiles. A group of 36 tiles has $5.4mm$ by $5.4mm$ dimensions (Figure 3.4a). The longest local link between the tiles and router is $2.7mm$. The links between routers are $5.4mm$ long. The radix of each router is 40 and the router operates at frequency of $2.2GHz$. The network diameter reduces from 46 hops to 6 hops when

compared to mesh.

From our studies, we find that concentrated meshes provide significantly lower throughput than mesh. This is because concentrated meshes have lower bandwidth and the inter-router links become a bottleneck. The local links between the tiles and cores seldom become the bottleneck. Thus, we consider a new concentrated mesh design which has multiple parallel links between routers to improve throughput. However, these additional links further increase the switch radix, and hence reduce the router frequency. Figure 3.4c shows the layout of a 36-degree concentrated mesh with 4 parallel links between the routers. The radix of each router increases to 52 and its frequency reduces to $1.8GHz$.

In our evaluations, we show that the conflicting trade-offs discussed above limit the benefits of concentration.

3.3.1.2 Flattened Butterfly

The flattened butterfly is a cost-efficient topology that can be extended to high-radix routers [59]. It is derived by combining the routers in each stage of a conventional multi-stage butterfly network. The flattened butterfly reduces hop count over conventional mesh by concentration as well as rich connectivity by using longer express links between non-adjacent routers.

The flattened butterfly topology can be scaled up by either increasing concentration, or increasing the dimensions (i.e. stages). For our studies, we choose to increase concentration. We limit ourselves to 2-dimensional flattened butterfly to reduce the stages and hence achieve a low network diameter of 2 hops. Also, the 2-dimensional flattened butterfly renders well to a 2D-planar layout.

The flattened butterfly uses symmetric high-radix routers, concentration, and express channels to improve scalability. Its symmetric nature trades off efficiency of local communication to achieve faster global communication. Also, its scalability in

terms of network throughput is limited due to concentration.

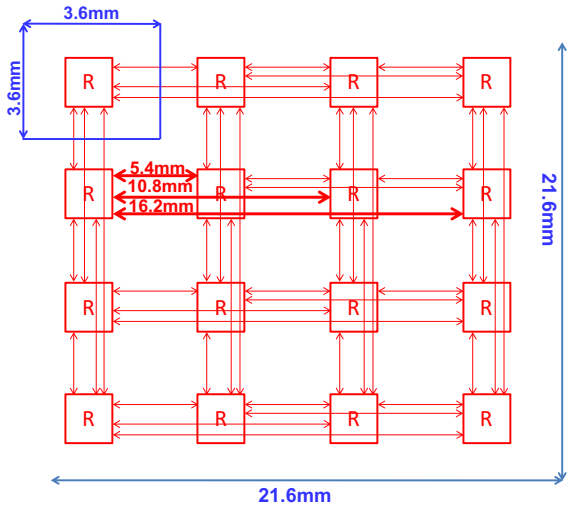


Figure 3.5: Flattened Butterfly Topology

Figure 3.5 shows the layout of the 4-ary 3-flat 2-dimensional flattened butterfly used in our studies. Each router is shared by 36 tiles. The cluster of tiles around a router will be similar to Figure 3.4a. There are 16 routers of radix-42 operating at a speed of $2.1GHz$. The longest link in the topology is about $17.6mm$ and is pipelined to deliver flits in 3 cycles.

3.3.2 Asymmetric High-Radix Designs

Above, we observed that traditional symmetric high-radix topologies trade-off local communication for global communication. These topologies have large high-radix routers which reduce hop count and optimize for global communication delay. But this is at the cost of higher local communication delay, which requires routing through the slow high-radix routers even for close-by cores.

Our approach towards designing a high-radix topology consists of three key elements. *First*, we split the communication into local traffic between cores which are near-by and global traffic between cores that are spread apart. This is not a new concept and has been used in prior interconnect designs [32] and in other contexts,

such as road systems in cities, power supply grids, etc. *Second*, we make the key observation that for each type of communication, router speed should match wire speed. For local communication—where cores are close-by, wires are short, and wire delay is small—the router should be fast and have lower radix. Since communication is local, the lower radix does not increase hop count significantly. For global communication the routes will be inherently long and wire latency will be large regardless of the number of pipeline stages. Hence, global routers can afford to be slower allowing their radix to be increased. With higher radix, the number of hops is reduced, which results in lower network latency for global communication. *Finally*, we tackle the problem of reduced network throughput in highly concentrated topologies by replicating the global routers.

Based on the above guidelines we explore two high-radix topologies: *Super-Star* and *Super-StarX*. As a comparison point, we also consider a third asymmetric high-radix topology that does not follow our design principle, *Super-Ring*, which employs the popular ring interconnect for global routers.

Multi-stage topologies such as trees [17, 68] and Clos [57] that have been proposed for on-chip networks have hop-counts proportional to the number of stages. The scalability of *Swizzle-Switch* to higher radices enables us to achieve optimal performance and power with only two-stages, thus precluding the need to explore greater than two-stage switches.

3.3.2.1 *Super-Star*

The first asymmetric design is a hierarchical star topology. In *Super-Star*, a cluster of nodes are connected to a fast medium-radix local router as shown in Figure 3.6a. Figure 3.6b shows the logical sketch of Super-Star with local routers and a global router. The global router is connected to all local routers. The network diameter is two hops. The number of global routers can be increased to provide higher through-

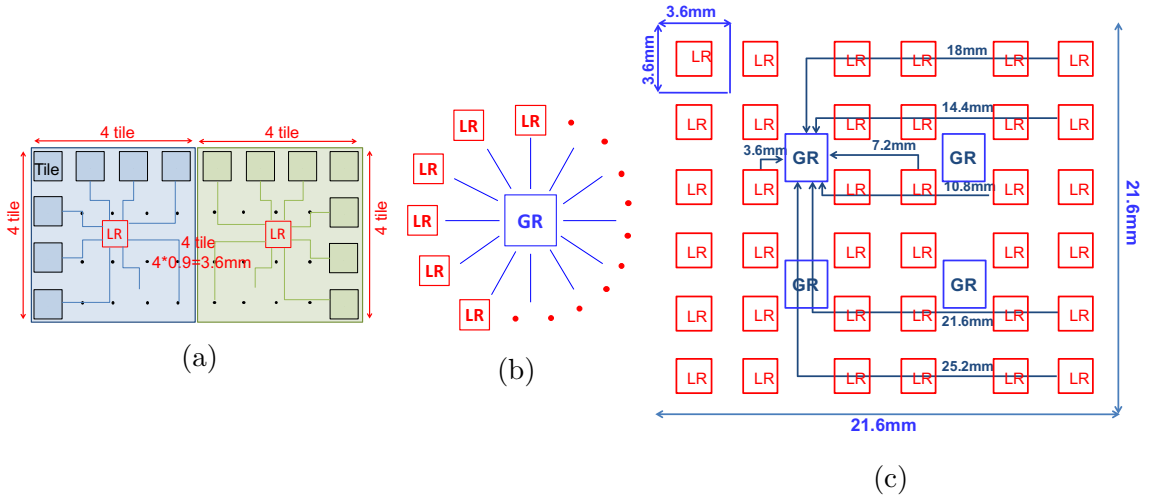


Figure 3.6: *Super-Star* Topology: (a) Layout of tiles within a cluster with a Local Router (LR). (b) Logical view of *Super-Star* showing connectivity between Local Routers (LR) and Global Router (GR). (c) Layout of *Super-Star* with four GRs.

put. There is no connection between the global routers. With multiple global routers, the *Super-Star* topology has the same topology connections as a 3-stage folded-Clos. However, current on-chip implementations of folded-Clos use equal radix routers [57]. This work is different in that we use few high-radix global routers and many low-radix local routers.

Figure 3.6c shows the physical layout of the Super-Star topology with 4 global radix-36 routers and 36 local radix-20 routers. The figure shows only a few distinct links with their dimensions for clarity. All outgoing links are pipelined to match the clock frequency of the router. Note, some global routers are spatially closer to a local router than others. However for simplicity and load balance, the global routers are chosen in a round-robin manner during the routing stage. More sophisticated routing schemes which account for wire-dimensions and buffer occupancy are also possible.

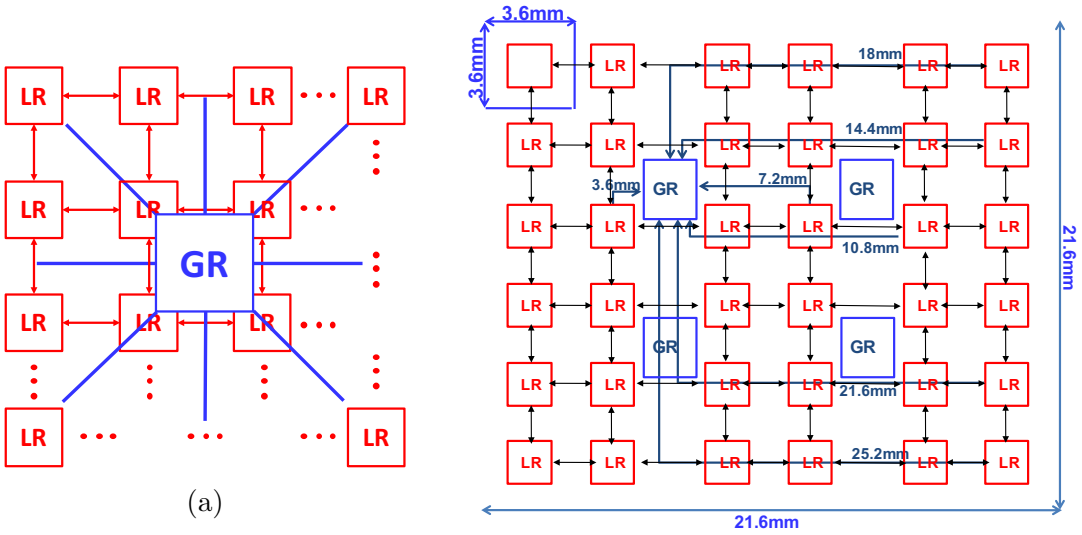
An interesting property of *Super-Star* is *energy proportionality*. The network throughput achieved by Super-Star topology and its power consumption is proportional to the number of global routers. Moreover, the entire network remains *fully connected* even with a single global router. Thus, network architects can choose to

have fewer global routers, if they are power constrained. Alternatively, the network can have a sufficient number of global routers to satisfy the peak throughput requirement. But when network load is low, a subset of global routers can be power-gated. In mesh and traditional symmetric high-radix topologies, energy proportionality is hard to achieve because *all* routers need to be active to keep the entire network fully connected, even when the overall network load is low.

3.3.2.2 *Super-StarX*

In the *Super-Star* topology, fast local communication is restricted to the cores within a cluster connected by the local routers. The local routers which are spatially close (i.e. neighbors in the layout) still need to communicate via a global router. We observe that providing connectivity between neighbors is cheap in terms of radix (the local routers radix only increases by 4, leading to minimal decrease in frequency), and this connectivity can reduce the latency of the local communication further. We refer to this new topology, which is derived from *Super-Star* by connecting the adjacent local routers as *Super-StarX*.

Figure 3.7a shows the logical sketch of *Super-StarX*. Note, all the beneficial characteristics of *Super-Star*, such as low latency, energy proportionality, etc, are preserved in *Super-StarX*. Although, sophisticated adaptive routing solutions are possible due to path diversity, we chose to implement a simple routing scheme in *Super-StarX*. The new links added between local routers are used only to communicate between neighboring local routers. All other inter-cluster communication between local routers is via the global routers. Thus, unlike concentrated mesh, in *Super-StarX*, the maximum number of hops is still limited to two hops. Figure 3.7b shows the layout of a *Super-StarX* topology with 4 global radix-36 routers and 36 local radix-24 routers (each local router is shared by 16 tiles). The link dimensions remain similar to *Super-Star* topology.

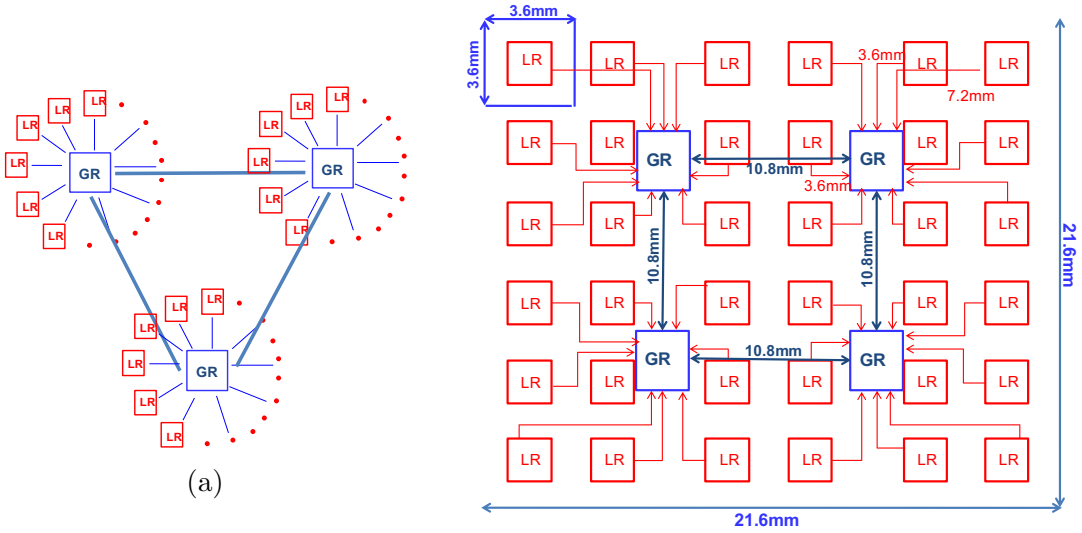


(a) (b)

Figure 3.7: *Super-StarX* Topology: (a) Logical view of *Super-StarX* showing connectivity between Local Routers (LR) and Global Router (GR). (b) Layout of *Super-StarX* with four GRs.

3.3.2.3 *Super-Ring*

Our previous asymmetric high-radix topologies (i.e. *Super-Star* and *Super-StarX*) connect the global router to all local routers. The local routers are medium-radix, fast, and matched to local wire delay. The global routers are high-radix, slower, and matched to global wire delay. Finally, we explore a topology which does not follow our design philosophy. In *Super-Ring*, the chip is divided into four logical quadrants with one global router per quadrant. The local routers are still medium-radix and match local wire delay. However, global routers are also medium-radix and are only connected to a subset of local routers. To provide full network connectivity, global routers are connected to each other in a ring. Note, all global routers need to be active for full connectivity, thus this topology is not energy proportional. Figure 3.8a shows the logical sketch of *Super-Ring*. Figure 3.8b shows the layout of a *Super-Ring* topology with 36 local radix-17 routers (each local router is shared by 16 tiles) and 4 radix-11 global routers. The link dimensions between local and global routers are



(a) (b)

Figure 3.8: *Super-Ring* Topology: (a) Logical view of *Super-Ring* showing connectivity between Local Routers (LR) and Global Router (GRs). (b) Layout of *Super-Ring* with four GRs.

shorter than *Super-Star* topology.

3.4 Evaluation Methodology

3.4.1 Router Delay and Power Model

We analyze the power and delay of each component of a router such as, links, buffers and switch (i.e. *Swizzle-Switch*), through SPICE modeling in 32nm industrial process and scale it conservatively to 15nm technology. Our models include energy spent due to clocking and leakage energy. The Swizzle-Switch architecture has been validated with a fabricated and tested silicon prototype [94, 96]. We assume a 128-bit Swizzle-Switch for all routers in our topologies and determine its frequency and power consumption at different radices. For each router, we assume a buffering of 4 virtual channels per port and a buffer depth of 5 flits per virtual channel. The routers utilize simple dual clock I/O buffer design with independent read and write clocks (similar to [73]). We conducted buffer sensitivity studies which showed that this much of

buffering was sufficient, even for topologies with long links. Our simulations model in detail the interface between routers operating at different frequency and multi-cycle links.

3.4.2 Link Delay and Power Model

Wire delays were determined using wire models from the design kit using SPICE modeling. Our analysis takes into account cross-coupling capacitance of neighboring wires and metal layers. For all links, we consider options that trade off energy for speed. We use different metal layers with either single or double spacing. Repeater insertion is adjusted so that repeaters are placed in the gaps between cores. The repeater placement was considered for all topologies to accurately estimate timing. On average the wire delay was found to be $66ps/mm$ and wire energy was found to be $0.07pJ/mm/bit$.

3.4.3 Performance Simulations

We use a cycle-accurate network-on-chip simulator for our analysis. All routers, irrespective of radix, use a two-stage microarchitecture [82]. We use simple deterministic routing algorithms, finite input buffering, wormhole switching, and virtual-channel flow control. The long links in different topologies were pipelined at the router frequency. The heterogeneity of frequency between routers was faithfully modeled. The activity factor of links, buffers and switches were collected from cycle-accurate simulations and integrated with power models to determine the network power.

We evaluate the proposed topologies with *uniform random statistical traffic* with a packet size of 512 bits (i.e. 4 flits). The datapath width is constant across all topologies and is equal to 128 bits. The network latency is reported is *nanoseconds* and the network throughput is reported in *packets/nanosecond/node*.

For applications, we use a trace-driven, cycle-accurate manycore simulator with

the above network model integrated with core, cache and memory controller models. Note, all the different components are tightly integrated to create a close-loop simulation environment. For example, the cores stall on a cache miss, the dependency between different coherence messages is obeyed, and queueing delays at the cache controllers and memory controllers are modeled. Thus, we can measure the execution time for the different workloads we simulate. Table 3.1 provides the configuration details.

Table 3.1: Simulated kilo-core processor configuration

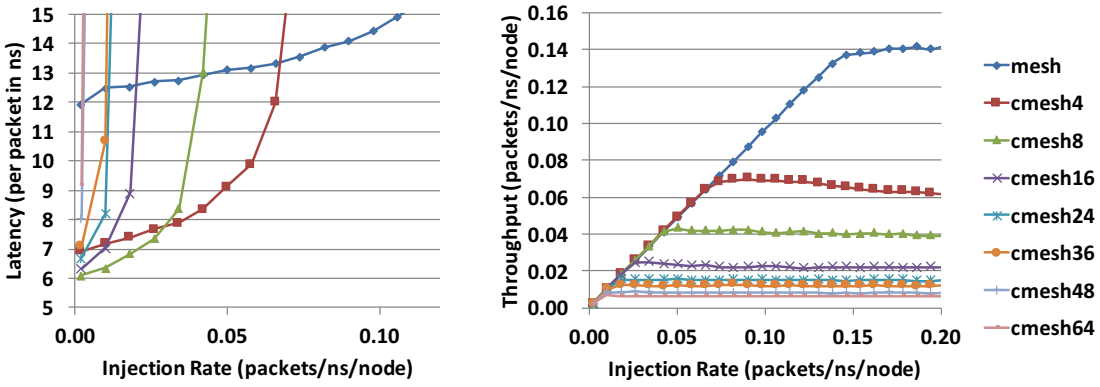
Cores	552 cores, 2-way out-of-order, 1GHz frequency
L1 Caches	32 KB per-core, private, 4-way set associative, 64B block size, 2-cycle latency, split I/D caches, 32 MSHRs
L2 Caches	552 banks, 256KB per bank, shared, 16-way set associative, 64B block size, 6-cycle latency, 32 MSHRs
Main Memory	24 on-chip memory controllers with 4 DDR channels each @ 16GB/s, up to 16 outstanding requests per core, 80ns access latency

We use a set of multiprogrammed application workloads comprising scientific, commercial, and desktop applications. In total, we study 44 benchmarks, including SPEC CPU2006 benchmarks, applications from SPLASH-2 benchmark suites, and four commercial workloads traces (sap, tpcw, sjbb, sjas). The traces for SPEC CPU2006 where collected using dynamic binary instrumentation [81]. The commercial workload traces where collected over Intel servers. The traces for SPLASH2 benchmarks were collected by running the benchmarks on gem5 full-system simulator [20]. The details of how each multiprogrammed workload mix is derived from the different single-threaded and multi-threaded benchmarks are discussed in Section 3.5.3.

3.5 Results

3.5.1 Analysis with Uniform Random Statistical Traffic

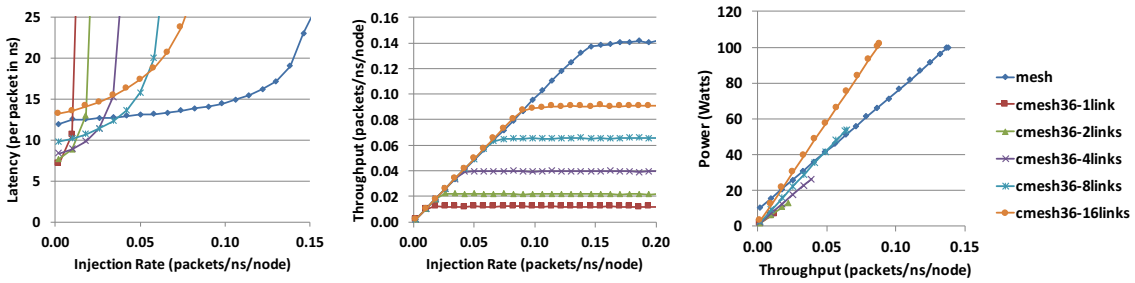
We first study the benefits and limitations of concentration. Figure 3.9a shows the average network latency and Figure 3.9b shows the network throughput with varying degrees of concentration. As postulated in Section 3.3.1.1, concentration provides excellent latency benefits at the cost of reduced throughput. Also, the latency benefits flatten out after reaching a concentration degree of 36. Beyond this concentration degree the benefits due to reduced hop count is countered by reduced router frequency. The average network latency before saturation drops from 16.8ns in mesh to 8.9ns for concentration degree of 36 and increases back to 9.2ns at a concentration degree of 64.



(a) (b)

Figure 3.9: Network latency (a) and throughput (b) for concentrated mesh with different concentration degrees.

In order to regain the throughput lost by concentration, we experiment with a new concentrated mesh topology with multiple parallel links. For this study we choose the largest concentration degree which provides the best latency and consumes the least power, i.e., concentration degree of 36. Although concentration degree of 8 has the best latency in Figure 3.9a, the higher number of routers dissipates more power. We maximize the concentration degree to reduce the number of routers and hence reduce



(a) (b) (c)

Figure 3.10: Network latency (a) throughput (b) and power (c) for concentrated mesh with additional number of inter-router links.

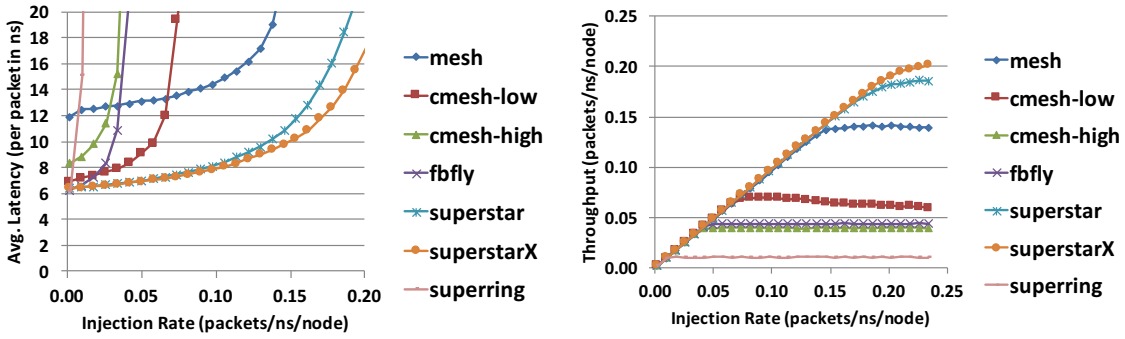
power. Figure 3.10b shows the average network throughput and Figure 3.10c shows the network power as a function of achieved throughput, with varying number of interrouter links. It can be seen that although we regain some of the lost throughput by adding additional inter-router links, the power grows steeply with additional links. Each additional set of links make the router bigger (router's radix increases by 4 times the number of parallel links), slower, and increases its power. The concentrated mesh with 16 parallel links consumes a power of $100.1W$ while providing a peak throughput which is only 60% of mesh's peak throughput. Thus, we conclude that concentration alone cannot scale the interconnect to kilo-core processors.

Table 3.2: Router radix, link dimensions, and network area for different topologies.

	# Routers		Radix		Network	Avg. Link Length(mm)	
Topology	Local	Global	Local	Global	**Area**	Local	Global
mesh	576	-	5	-	38.19	0.79	-
cmesh-low	144	-	8	-	13.18	1.28	-
cmesh-high	16	-	52	-	15.20	3.25	-
fbfly	16	-	42	-	10.82	3.56	-
superstar	36	8	24	36	18.24	1.80	12.90
superstarX	36	8	28	36	21.45	2.11	11.30
superring	36	4	17	11	7.12	1.80	6.48

Next, we study the different asymmetric high-radix topologies. We present the best configurations of each topology. Table 3.2 provides the number of routers and their radix, network area and link dimensions for different topologies. The design

goal was to restrict interconnect to 5% of chip area ($466mm^2$) while meeting performance and power targets. Figure 3.11 shows the average network latency and network throughput for different topologies. The low-radix *mesh* topology has high average network latency because of large number of hops. However, it is also able to achieve good network throughput because it has high bandwidth. The average latency of *mesh* topology before saturation is $16.8ns$ and it saturates at the throughput of $0.14packets/ns/node$.



(a) (b)

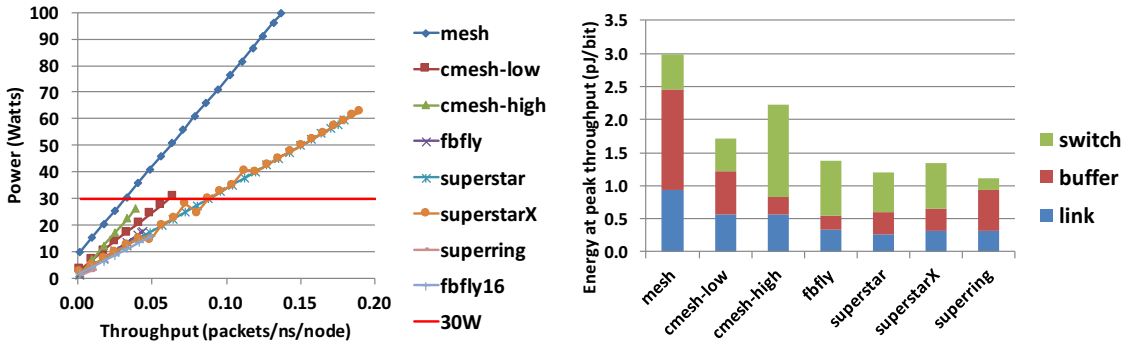
Figure 3.11: Performance of different topologies with uniform traffic:(a) Network latency and (b) Network throughput.

In contrast, the symmetric high-radix topologies enjoy low latency because of reduced hop count. However, they quickly saturate because of bandwidth bottleneck in inter-router links. The *cmesh-low* topology has a low concentration degree of 4. The *cmesh-high* topology has a high concentration degree of 36 and in addition has 4 parallel links between the routers. The *cmesh-low*, *cmesh-high* and flattened butterfly (*fbfly*) topologies have an average network latency of $9.6ns$, $10.8ns$ and $7.9ns$ and a saturation throughput of $0.07packets/ns/node$, $0.04packets/ns/node$ and $0.044packets/ns/node$. We also studied improving the bandwidth of symmetric high-radix topologies by increasing the datapath width and link width beyond 128 bits. However, we find that increasing datapath width makes the router slower as well as increases network power consumption significantly.

The asymmetric high-radix *Super-Star* and *Super-StarX* topologies enjoy both low latency and high throughput. They achieve low latency by effectively optimizing both local and global communication. They achieve high network throughput by having multiple global routers. The *Super-Star* and *Super-StarX* topologies have an average network latency of $9.3ns$ and $9.5ns$, about 45% improvement over *mesh*. Note, since we are simulating uniform random traffic pattern, the *Super-StarX* latency is similar to *Super-Star*. As shown later in a clustered traffic study, *Super-StarX* provides better latency for higher proportion of local traffic. Again, all average latencies are taken before saturation. While *fbfly* has a lower average latency than these topologies, it also saturates at a lower throughput. The *Super-Star* and *Super-StarX* topologies have a saturation throughput of $0.18packets/ns/node$ and $0.20packets/ns/node$.

The *Super-Ring* topology, although an asymmetric high-radix topology, was designed without adhering to our goal of matching router delay to wire delay. In this topology, the global routers are medium-radix, smaller and faster. Thus, global wire-delay is not matched to router speed. The local routers are still medium-radix. In addition, there is no redundancy between the global routers. Thus, inter-router links between global routers can become bandwidth bottlenecks. The *Super-Ring* provides an average latency of $10.6ns$ and quickly saturates at throughput of $0.01packets/ns/node$. We conclude that matching wire delay with router speed is important and a naive asymmetric hierarchical topology cannot provide optimal performance.

Figure 3.12 shows the network power and network energy for different topologies. Figure 3.12a plots the network power (Y-axis) as function of achieved network throughput (X-axis). The network power increases with increasing network throughput. The lines for different topologies stop at different throughputs and the end points correspond to the saturation throughput of that topology. It can be seen that symmetric high-radix topologies stop very quickly due to their lower throughput. In



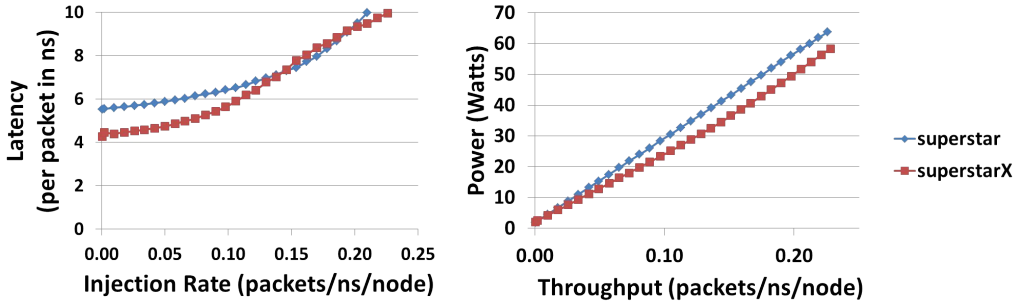
(a) (b)

Figure 3.12: Power characteristics of different topologies with uniform traffic:(a) Network power and (b) Network energy.

general, the topologies which reach further right and have a slow slope of increase in power with respect to throughput are more desirable. It can be seen that *Super-Star* and *Super-StarX* topologies achieve the best power efficiency: 1) their slope of power increase with respect to throughput is smallest and 2) their achievable throughout is farthest to the right. They can achieve 39% higher throughput while consuming only 60% of power when compared to *mesh*. If we limit the network power to $30W$ across all topologies, the proposed *Super-Star* and *Super-StarX* topologies can provide 3× higher throughput than mesh and 1.4× higher throughput than *cmesh-low*. Figure 3.12b shows the energy per bit of the different topologies at a low injection rate of $0.04 packets/ns/node$. It can be seen that the proposed high-radix topologies trade-off link and buffer energy for switching energy.

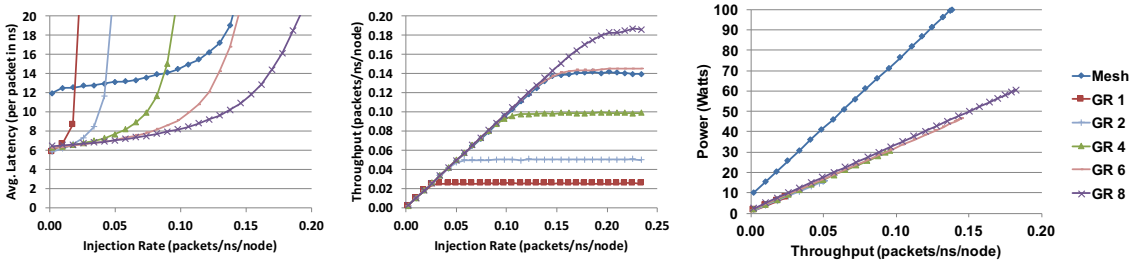
To further emphasize the benefit of providing fast connectivity to adjacent local routers in *Super-StarX*, we simulated a clustered traffic pattern. In this traffic pattern, communication is only to cores within the same cluster or to cores in adjacent clusters. The cluster size of both *Super-Star* and *Super-StarX* is 16 tiles. The locality-aware routing policy of *Super-StarX* uses the links between local routers to route most packets. The routing policy adapts to high congestion by routing packets via the global routers when the buffer occupancy for links between local routers exceed a

predetermined threshold. Figure 3.13 shows the network latency and network power for this study. The additional connectivity and the adaptive, locality-aware routing policy of *Super-StarX* provide much lower latency than *Super-Star* and better power efficiency.



(a) (b)

Figure 3.13: Network latency (a) and Network power (b) for clustered traffic study



(a) (b) (c)

Figure 3.14: Energy proportionality of Super-Star topology with varying number of global routers:(a) Network latency (b) Network throughput and (c) Network power.

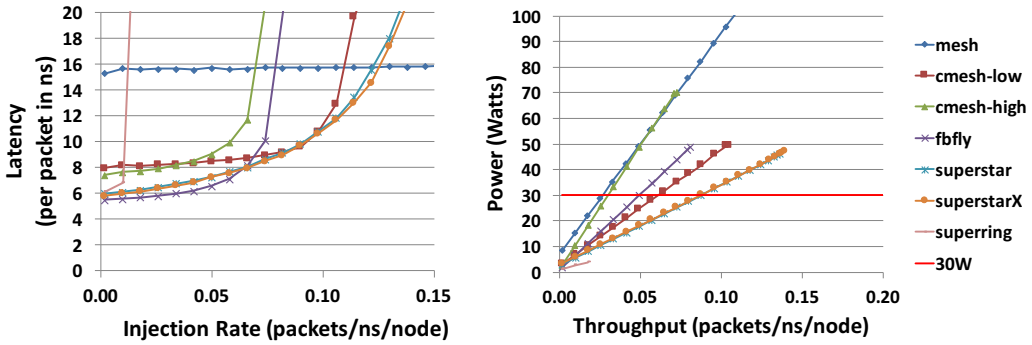
Finally, we evaluate the energy proportionality of our proposed *Super-Star* topology. Figure 3.14b shows the proportional growth in throughput as we increase the global routers (GRs) from 1 to 8. Figure 3.14c shows that the network power increase with respect to achieved throughput has similar slope for all the different configurations (GR1 to GR8). Thus, if the required throughput of the system is low, designers can save power by using fewer GRs. In mesh and symmetric high-radix topologies, all routers are necessary to provide full network connectivity. Thus, it is hard to design these networks in an energy proportional manner. To bound these topolo-

gies to a lower Thermal Design Power (TDP) budget (e.g. TDP is equal to $30W$), they will have to be either 1) under-clocked, sacrificing latency or 2) have complex source throttling mechanisms to limit the injection rates at source nodes such that the network power does not exceed the pre-decided TDP.

3.5.2 Bisection Bandwidth Wires

Table 3.3: Bisection bandwidth wires of different topologies for equal wires study.

Non-equal # Wires		**Equal # Wires**	
Bus Width	Bisection Wires	Bus Width	Bisection Wires
128	6,144	512	24,576
128	9,216	256	18,432
128	10,240	256	20,480
128	10,240	256	20,480
128	36,864	73	21,024
128	44,544	64	22,272
128	9,216	256	18,432



(a) (b)

Figure 3.15: Network latency (a) and Network power (b) for equal wires study

Bisection bandwidth of the topologies in Figure 3.11 varies significantly. We assumed a constant 128-bit datapath width, which results in different number of wires at the bisection for different topologies, as listed in Table 3.3. The bisection wires include wires from tiles to local routers as well as inter-router links. For a better comparison of topologies, we conducted a new study with an equal number of wires

for all topologies. To achieve the same number of wires (approximately 21,000), the datapath width was adjusted according to number of links at bisection. The new channel widths are listed in Table 3.3. Figure 3.15 shows the average network latency and network power for this study. The wider datapath of mesh and *cmesh-low* causes the frequency of the router to decrease, thus we observe a small increase in latency compared to Figure 3.11. Except for *cmesh-high* and *fbfly*, which benefits more from the additional bandwidth than the loss due to decreased router frequency. On the other hand, the narrower datapath of *Super-Star* and *Super-StarX* causes their throughput to saturate at a lower injection rate. Routers of *Super-Star* and *Super-StarX* become smaller due to the narrower channels, which results in better power efficiency, whereas the large channels of *mesh* and *cmesh-high* increases switch power significantly. Similar to Figure 3.12a, if we limit the network power to $30W$ across all topologies, the proposed *Super-Star* and *Super-StarX* topologies can provide 3× higher throughput than mesh and 1.4× higher throughput than *cmesh-low*.

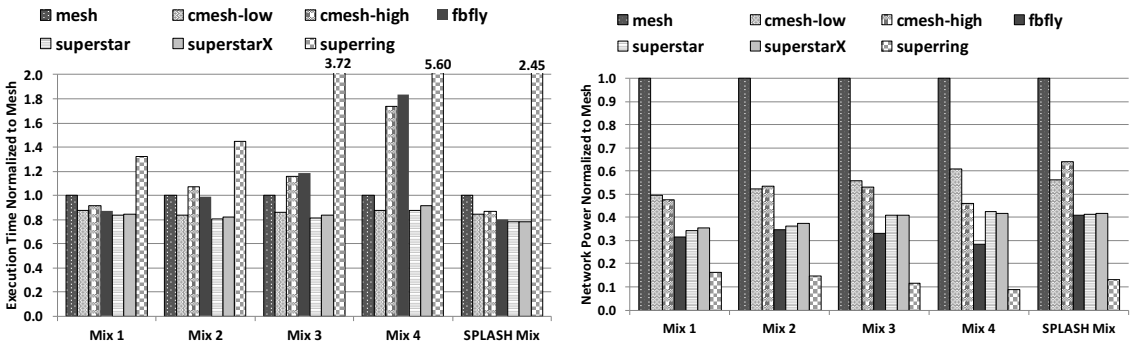
3.5.3 Application Workloads

Table 3.4: List of workloads with their cache miss rates.

Workload	**Applications**	**L1 MPKI**	**L2 MPKI**
Mix 1	applu, astar, barnes, bzip2, calculix, gcc, gobmk, gromacs, hmmer, perlbench, sjeng, wrf	2,543	807
Mix 2	applu, bzip2, calculix, deal, gcc, gromacs, libquantum, perlbench, sap, sjeng, tonto, wrf	4,173	1,854
Mix 3	art, calculix, gobmk, gromacs, h264ref, libquantum, namd, ocean, omnet, perlbench, sap, sjas	7,211	3,119
Mix 4	astar, deal, Gems, gobmk, gromacs, lbm, leslie, mcf, milc, namd, sjeng, swim	15,899	9,263
SPLASH mix	Barnes, Cholesky, FFT, FMM, Lu, Ocean, Radix, Raytrace	4,096	1,408

In this section, we study the characteristics of different topologies with real application workloads. We evaluate five multiprogrammed workloads. The first four workloads, *Mix 1*, *Mix 2*, *Mix 3* and *Mix 4*, run 46 copies of 12 unique applications which are chosen randomly from our suite of 35 single-threaded applications. The

fifth workload, *SPLASH mix*, runs 64 threads each of 8 parallel applications taken from SPLASH-2 benchmark suite. The workloads are listed in Table 3.4 along with the total cache miss rate of each workload measured in terms of Misses Per Kilo Instructions (MPKI).



(a) (b)

Figure 3.16: Performance of different topologies with application workloads: (a) Execution Time (a) and Network Power

Figure 3.16a shows the system performance of various topologies and Figure 3.16b shows the network power consumption. We can observe that trends from our studies with statistical traffic persist. The *Super-StarX* topology provides an average performance improvement of 17% over the mesh topology while consuming 39% less power. Although the symmetric high-radix topologies and *Super-Ring* topology consume lower power, they have higher degradation in performance because they provide lower network throughput. The asymmetric high-radix topologies both improve performance and consumes lower power

3.6 Related Work

In this chapter, we study the scalability aspects of switch design and network topology design in the context of kilo-core processors. Below we summarize the closely related works.

3.6.1 Network-on-Chip Topologies

Todays multicore processors use a variety of interconnect topologies such as shared bus, rings, crossbars and meshes. The shared bus fabric was the prevalent interconnect design for decades because of low design complexity, low power consumption, and ability to support snoop-based coherence protocols. Unfortunately, buses do not scale beyond a few cores. Kumar et al. [64] showed that the shared bus fabric does not scale beyond 16 cores. To overcome scalability limitations, multicore processors adopted crossbars and rings.

The Niagara processor [63] implemented a crossbar interconnect to facilitate communication between 8 cores, 4 cache banks and I/O modules. Niagaras interconnect consisted of two 124b and 145b crossbars, operating at $1.2GHz$ frequency in 65nm technology, providing a data bandwidth of $134.4GB/s$, while consuming ~$3.8W$ of power. Recently, IBM Cyclops64 [123] supercomputer manycore processor chip implemented a 96-radix, 96b wide crossbar operating at $533MHz$ and occupying an area of $27mm^2$.

Ring interconnects have been popular with multicore processors [14, 46, 102] due to relative simplicity of design of individual switches and the ability to provide global ordering. IBM Cell [14] has four 128b unidirectional rings operating at $1.6GHz$ frequency and supporting data bandwidth of ~$200GB/s$. STs Spidergon [22] proposes a bidirectional ring augmented with links that directly connect nodes opposite to each other on the ring. These additional links reduce the average hop distance. To overcome bandwidth limitations, recent ring implementations use wide datapaths (e.g. Intels Sandybridge [105] processors use 256b rings). Unfortunately, rings bisection bandwidth does not scale with the number of nodes in the network, limiting it scalability to few dozens of cores.

The 2D mesh [7, 48, 91, 111, 119] topologies have become popular for tiled manycore processors because of their low complexity, planar 2D-layout properties and

better scalability compared to rings. The TILE64 processor [119] implements five 32b 8×8 mesh networks to support various message classes and connect 64 nodes. Intels Single Cloud Computing (SCC) [48] processor chip implements a 128b 6×4 concentrated mesh interconnect where two cores and two cache tiles share a router. SCCs interconnect consumes ~12W power while operating at 2GHz frequency in 45nm technology.

Beyond commercial processor implementations, on-chip network topologies have been explored actively by academic researchers. Wang et al. [118] did a technology oriented, and power aware topology exploration for mesh/tori topologies with analytical models.

Several designs have been proposed to overcome the inefficiencies of 2D-meshes. Hierarchical bus-based topologies [32, 113] have been proposed to reduce power consumption and minimize network latency. The bus-based proposals have limited scalability and were optimized for processors with 32 to 64 cores. Balfour and Dally proposed concentrated meshes [8] with express channels. Kim et al. [59] proposed flattened butterfly topology to reduce latency by providing rich connectivity.

Grot et al. [44] proposed multi-drop express channels (MECS) to reduce network latency by facilitating one-to-many communication over long express channels. The multi-drop concept of MECS topologies can be applied to the long channels in our proposed topologies to further improve network latency. However, the MECS topology can have significant buffering requirements to cover credit round trip delays over the express channels [45], as we scale up the network size. In [45], the authors discuss the challenges of scaling on-chip networks towards 100s of cores and propose use of the MECS topology to reduce cost of providing quality-of-service in network-on-chips with up to 256 nodes.

We believe, that while the above proposals were good designs which improved network latency significantly over the mesh topology, the design challenges and trade-

offs for a kilocore processor interconnect are different. Our proposed designs leverage the rich diversity of radix offered by *Swizzle-Switches* and our design space exploration is guided by wire delay slack leading to asymmetric radix designs. In our evaluations, we analyze the scalability of existing symmetric radix topologies such as concentrated meshes and flattened butterfly and compare our proposed designs to them.

Multi-stage fat trees [68], Reduced Unidirectional Fat Trees (RUFT) [87] and Clos [57] topologies have been also considered for on-chip networks. However, these proposals were based on traditional switch designs and thus limited all routers to radix-8. The Rigel 1000-core accelerator [54] proposes the use of a multi-stage tree interconnect.

In our design, routers with different radices operate at different frequencies. Prior work has exploited multiple frequency domains in 2D-mesh interconnects to manage congestion [73] and apply Dynamic Voltage Frequency Scaling (DVFS) [48].

Previously deployed systems have used a hybrid of multiple types of topologies to achieve high bandwidth across the entire computing system. The Thinking Machine Coporation's CM-2 combined mesh and hypercube by having each node in the hypercube be a mesh of simpler nodes [35]. Another design used the dragonfly topology to connect cabinets and a 3D flattened butterfly topology to connect the nodes inside the cabinet [61]. Hybrid topologies exhibit a degree of asymmetry in which the topology at each level is selected to optimize the type of communication at that level. In our *Super-StarX* design that superimposed a mesh on a folded-Clos, we introduce this type of asymmetry at the on-chip network level. We believe going forward kilo-core processors will continue to trend towards a combinations of topologies.

3.6.2 High-Radix Switches

Prior works have recognized the multifaceted benefits of high-radix switches [58, 59, 60, 101]. Kim et al. [58] proposed several optimizations to improve the scalabil-

ity of switches with respect to radix. The optimizations included breaking down the arbitration into multiple local and global stages, decoupling the input and output virtual channel/switch allocation by including intermediate buffers at cross points and hierarchical crossbars with intermediate buffering. Recently, Passas et al [79, 80] proposed high-radix crossbar interconnects for 128 tile chips. Their implementation of a 128-radix crossbar was 32b wide, divided the data transfer into 3-stages and operated at a frequency of $750MHz$ at 90nm technology. The crossbar datapath occupied an area of $7.6mm^2$, while the arbitration logic (or scheduler) is a iSLIP [69] scheduler and occupies an area of $7.2mm^2$. Their work recognizes that arbitration complexity is a bottleneck in designing high-radix switches and proposes wiring optimizations to reduce the arbitration delay to $10ns$.

In contrast to above decoupled approaches, *Swizzle-Switches* take an integrated approach towards arbitration to provide excellent scalability. The datapath and arbitration in a *Swizzle-Switch* is tightly coupled in a SRAM-like layout, reducing the area and critical path delay for the switch. Unlike traditional logic-tree arbiters, the arbitration in *Swizzle-Switch* is done by updating the internally stored priority bits on a cycle-by-cycle basis.

3.7 Summary

To realize kilo-core processors, it is important that we find a solution for designing a performance and power scalable on-chip interconnection network. In this chapter, we proposed a class of asymmetric high-radix topologies that decouple local and global communication optimizations. Our proposed topologies employ the design principle that routers need to be only as fast as the wires that connect them. Thus, we employed fast, medium-radix switches for local routers to achieve efficient local communication. Using a few high-radix global switches to connect local routers, we were able to reduce the hop count for global communication and also improve the

overall throughput of the network.

Our experiments demonstrated that the best performing asymmetric high-radix topology improves average network latency over mesh by 45% while reducing the power consumption by 40%. When compared to symmetric high-radix topologies (i.e. concentrated meshes and flattened butterfly) our proposed topologies improve network throughput by 2.9× and network latency by 14% while providing similar power efficiency.

CHAPTER IV

Hybrid Checkpointing to DRAM and SSD

4.1 Introduction

Aggregate failure rates of millions of components result in frequent failures in exascale supercomputers. In particular, exascale systems are projected to have memory systems as large as 100 petabytes—that is 100× larger than the supercomputer Titan's 1 petabyte memory system. The millions of memory devices that make up these memory systems contribute significantly to failures [100] and overcoming them requires a fast and reliable checkpoint/restart framework.

Checkpointing—periodically saving a snapshot of memory to stable storage—is a useful practice to rollback the application to a point before failure, without restarting from the very beginning. Exascale systems rely heavily on checkpoints to recover from many types of failures including hardware failures, software failures, environmental problems, and even human errors [99]. Usually, checkpoints are made to a non-volatile storage such as a hard disk, but increasingly, solid-state drives (SSDs) are replacing hard disks because they provide higher read/write bandwidth, lower power consumption, and better durability [78]. The question becomes whether SSDs are sufficient for storing checkpoints or if we should wait for emerging memory technologies.

The biggest disadvantage of NAND-flash SSDs is its lower endurance, which is on the order of 10^4-10^5 program/erase cycles. SSD manufacturers employ various tricks

such as DRAM buffers and sophisticated wear-leveling to extend lifetime. Currently, SSDs on the market are guaranteed a lifetime of 3-5 years with a cap on the total number of terabytes that can be written [51]. Nevertheless, writing gigabyte-sized checkpoints several times a day to the SSD can take a toll on its endurance.

Many have suggested using emerging non-volatile memory technologies such as phase change memory, memristors, and STT-RAM for checkpointing, often touting their superior read and write speeds and higher endurance [29, 34, 56]. While we do not disagree with these studies, emerging technologies must overcome many undeveloped steps between a successful prototype and volume production. It is difficult to guess when, or if ever, emerging technologies will be ready for the first round of exascale supercomputers. The U.S. Department of Energy's Exascale Computing Initiative plans to deploy exascale computing platforms by 2023 [116]. Designs for 2023 systems will have to be finalized 3-4 years prior, similar to plans for Summit (2018) and Aurora (2018-2019) supercomputers that were completed by 2015. At some point, system designers will have to reason about reliable, off-the-shelf components that will be available in the next 3 years. We show that existing non-volatile storage options that are proven less risky due their maturity and low cost are sufficient for the near future, if used correctly.

When using SSD flash memory for checkpointing, reducing the checkpoint size or frequency remain the most effective ways to stretch its lifetime. To this end, we propose a system that selectively checkpoints to a DRAM in order to reduce the number of writes to the SSD thereby lengthening its useful lifetime. To accomplish this task, we implement a Checkpoint Location Controller (CLC) that i) estimates SSD lifetime, ii) estimates application's performance loss, and iii) monitors checkpoint size. The CLC detects checkpointing frequencies that lead to SSD lifetime falling under the typical manufacturer's guarantee of 5 years, and reduces these frequencies by redirecting some checkpoints to the DRAM. We believe this is the first work to

consider the lifetime of the SSD while writing checkpoints to it; previous work [67] that also used SSD ignored its endurance.

DRAM is prone to transient errors and checkpoints corrupted by them cannot be used for recovery. Then, a key feature to enable our technique of writing fewer checkpoints to the SSD is to have a strong error correcting code (ECC) that can protect the checkpoints in DRAM. For that reason, we propose a dual mode ECC memory system that protects regular application data with a normal ECC algorithm and checkpoint data with a strong ECC algorithm. The normal ECC, which is on the critical path of memory accesses, is an RS(36,32) code that has small decoding latency to correct or detect errors. It can correct all errors due to a bit/pin/column/word failure and detect all errors due to a chip failure. The strong ECC is a two-layer RS(19,16) code that provides Chipkill-Correct level reliability without modifications to the DRAM devices. If an unrecoverable error corrupts the DRAM checkpoint, then the application will restart from the checkpoint in the SSD. The resultant capability to write reliable checkpoints to memory relieves the burden on the SSD, in turn lengthening its lifetime. More importantly, the combined DRAM-SSD checkpointing solution makes it possible to design an exascale memory system without relying on unproven emerging memory technologies.

In summary, we make the following contributions:

- **A low-risk exascale memory system.** We use mature technology in commodity DRAMs and SSDs to create a low design-risk checkpointing solution and prove that system designers do not have to wait until newer non-volatile memory technologies are ready.

- **Hybrid DRAM-SSD checkpointing**. Our local checkpointing solution is a hybrid mechanism that uses both DRAM and SSD flash memory to achieve speed and reliability (Section 4.3).

- **SSD-lifetime-aware checkpoint controller.** We design an intelligent Checkpoint Location Controller (CLC) that decides when to checkpoint to the SSD considering its endurance decay and performance degradation (Section 4.3.3).

- **Dual-ECC memory.** We propose a dual mode ECC memory that has a normal ECC mode to protect regular application data and a strong ECC mode to protect the DRAM checkpoint. ECC-protected checkpoints ensure error-free restarts at recovery (Section 4.4). The dual-ECC modes were developed in collaboration with researchers at Arizona State University.

Our results from microbenchmark simulations averaged across various checkpoint sizes indicate that the CLC is able to increase SSD lifetime by 2×—from 3 years to 6.3 years—exceeding the guaranteed lifetime of 5 years [51]. Furthermore, the performance estimation feature in the CLC that monitors application slowdown is able to reduce the checkpoint overhead to a 47% (on average) slowdown, compared to a 420% slowdown when the application always checkpointed to the SSD—nearly a 10× savings.

4.2 Motivation

Local checkpoints to local storage (DRAM or SSD) have stemmed from a need to avoid the slowdown resulting from transferring checkpoints to the remote parallel file system (PFS) over limited-capacity I/O channels. It is difficult to decide on the best local storage because each has their advantages and disadvantages. On one hand, DRAM is fast (50ns [117]) but loses the checkpoint after a reboot. Furthermore, limited DRAM capacity not only limits the size of the largest checkpoint that can be made but also limits the amount of usable memory for applications.

On the other hand, SSDs are reliable and capacious but slow and have low endurance (10^5 program/erase cycles). To illustrate the speed difference between

ramdisk—a virtual disk created in DRAM to write checkpoint files—and the SSD, we measured the total runtime of a microbenchmark (details provided in Section 4.5.1) under three naïve implementations i) no checkpointing, ii) checkpointing to ramdisk only, and iii) checkpointing to SSD only. For this simulation, we assumed that both ramdisk and the SSD had unlimited checkpoint storage. As can be seen in Figure 4.1, writing the checkpoint to ramdisk incurs a small 14% slowdown, but checkpointing to the SSD incurs a considerable 4.6× slowdown averaged across all the checkpoint sizes.

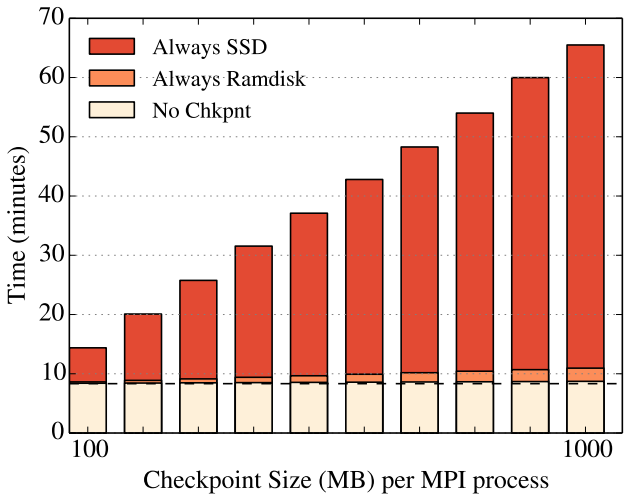


Figure 4.1: Microbenchmark runtime results with various checkpoint sizes demonstrate that always checkpointing to the SSD incurs significant overhead. Baseline runtime = 8.3 minutes.

A key observation that we made during our experiment was that even when checkpointing only to the SSD, files are first written into the page cache allocated in the main memory. Files in the page cache are not necessarily flushed to the storage device when the file is closed because the operating system chooses to delay writes to block devices in order to hide I/O latency. The operating system provides no guarantee as to when the checkpoint will be persisted. Waiting for the operating system to write back data at its own discretion puts the checkpoint in a vulnerable state, exposed to memory failures and power failures. On the other hand, the programmer can choose

to explicitly flush the page cache after each checkpoint at the cost of performance overhead because the slow write delay to flash becomes fully transparent to the application. Our solution to this dilemma is to write a select few checkpoints to the SSD and always flush them. To balance out the performance loss, we explicitly write the remaining checkpoints to main memory—not to the page cache, but rather to a *ramdisk* specifically for writing checkpoints. Explicitly writing checkpoints to the memory (as opposed to the letting the operating system implicitly write them), allows the application to know which of its checkpoints are not guaranteed to be safe.

The hybrid solution merges the benefits of both DRAM and SSD: namely, speed and reliability. Furthermore, checkpointing to the DRAM helps to reduce SSD wearout. The shortcomings of our solution is that it limits the available memory for applications and increases the memory pressure (i.e. ratio of active memory pages) due to active checkpoints residing in memory. The pros and cons of the proposed technique are listed in Table 4.1.

The checkpoints in ramdisk are exposed to DRAM failures, but ECC algorithms exists that are capable of protecting against most memory failures—except for a power outage. The stronger the ECC, the more time and power that it takes to decode data. A second key insight into our idea is that it is possible to use stronger ECC algorithms for checkpoints because decoding them is not on the critical path of normal application execution.

Alternative memory technologies such as phase-change, magnetic, resistive RAM, and 3D XPoint holds promise because they are almost as fast as DRAM (10-300ns [117]), yet also as reliable as storage. However, these technologies are not yet as dense or cost-efficient as flash. Although Intel's 3D XPoint is expected to cost half of DRAM [77], recent innovations in 3D NAND-flash such as stacking 48 layers [55] will only cheapen flash. Furthermore, unlike emerging technologies, flash devices have well understood failure patterns and strong ECC codes to protect them [110]. Commercial availabil-

Table 4.1: The pros and cons of the proposed technique compared to DRAM-only or SSD-only checkpointing. The memory occupancy is marked as "Med" because the CLC can detect and send large checkpoints always to the SSD.

	DRAM only	SSD only	Proposed DRAM+SSD
Checkpoint Speed	Hi	Lo	Hi
Recovery Speed	Hi	Lo	Hi
Transient error protection	Lo	Hi	Hi
Available memory for apps.	Lo	Hi	Med
Memory pressure	Hi	Lo	Med
Non-volatile; persists reboots	N	Y	Y
Good SSD endurance	–	N	Y

ity and maturity of both DRAM and NAND-flash prove them a low-risk option for at least the first round of exascale systems. Should emerging technologies become better than flash, they can easily be integrated into our hybrid system and achieve even better performance.

In the remainder of the chapter, we address two questions: 1) how to decide when to checkpoint to the DRAM or the SSD? and 2) how to design a strong ECC algorithm to protect the checkpoints without interference to non-checkpoint-data memory accesses? To answer the first question, we implement the CLC in Section 4.3.3 that is aware of the endurance limits of the local SSD device and the performance degradation from writing to it. To answer the second question, we introduce a dual-mode ECC design in Section 4.4 that can be dynamically encode data in either normal ECC or strong ECC depending on whether the data is normal application data or checkpoint data.

4.3 Hybrid DRAM-SSD Checkpointing

An overview of the hybrid solution is presented in Figure 4.2. Where past systems chose either DRAM or SSD as the checkpointing platform [89, 124], our hybrid

solution uses both considering SSD lifetime, application performance, and checkpoint size. In our system, all compute nodes contain main memory DIMMs consisting of x4 ECC-DRAM devices and one high-capacity SATA SSD with flash. Our system can exist within a hierarchical framework where global checkpoints are written to the remote PFS. Note that this system differs from *double checkpointing* in other work [76, 124] that write identical checkpoints to two platforms in "buddy" nodes. Double checkpointing wastes memory space. In contrast, the hybrid system writes only one checkpoint to one platform in a given checkpoint interval as illustrated in Figure 4.3. Although not implemented in this work, a possible optimization is to implement the hybrid system on top of a buddy system, where either the ramdisk or SSD checkpoint is saved in the buddy's ramdisk or SSD, respectively.

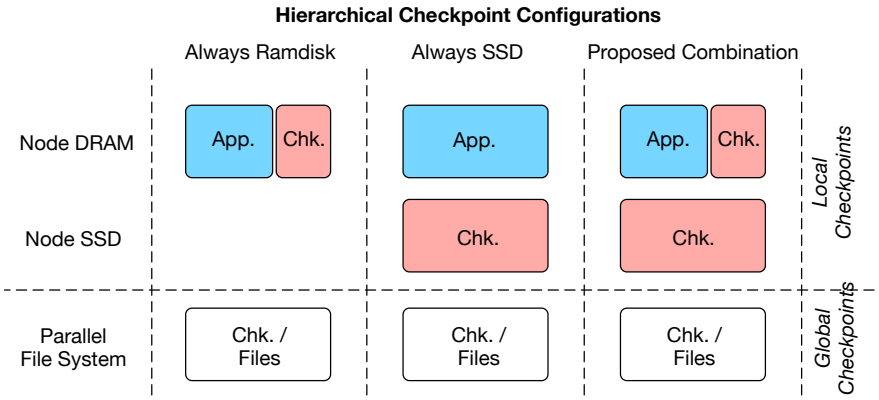


Figure 4.2: The proposed idea utilizes both commodity DRAM and commodity SSD for checkpoints.

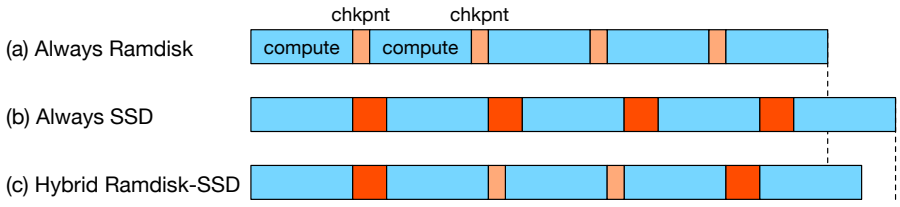


Figure 4.3: In the hybrid system (c), the CLC intelligently selects which checkpoints are to be written to the SSD considering endurance, performance, and checkpoint size.

4.3.1 Checkpointing to the Ramdisk

Checkpoints to memory are written outside of the application's address space to ensure its persistence after the application crashes or ends. This can be achieved by writing checkpoints to the *ramdisk*. There are two types of ramdisk file systems: *ramfs* and *tmpfs*. The main difference between them is that ramfs cannot be limited in size—i.e. it will keep growing until the system runs out of memory—whereas tmpfs will start swapping to disk once the specified size limit is full. We use tmpfs and enforce a size limit that ensures checkpoint memory does not encroach upon the application's memory.

4.3.1.1 Memory Requirement

In-memory checkpointing to DRAM requires prudent management of memory resources. Out of the available memory on each server node, a certain quantity is set aside for checkpointing by mounting a ramdisk into the memory space. The user should consider the memory requirement for both the application and the checkpoint. For example, 4GB out of a 24GB system can be set aside for checkpoints, leaving only 20GB for the application. The high performance application running on the node can be adjusted for the smaller memory size by setting a smaller problem size per MPI process, or by running fewer MPI processes on the node.

4.3.2 Checkpointing to the SSD

Since the future of emerging NVMs are unclear, we suggest that they should not be used in the first generation of exascale machines. Commercial availability and maturity of both DRAM and NAND-flash prove them a low-risk option sufficient for at least the first generation of exascale systems, if used correctly.

In fact, since flash SSDs are readily available in the market, HPC system designers are already considering them for checkpointing. SSDs have been most commonly

proposed as a "burst buffer"—a storage buffer that is placed between the compute nodes and storage nodes to quickly absorb bursty I/O traffic [67]. The original burst buffers paper suggested placing the SSDs in the I/O nodes because they still have system-wide visibility. Subsequent suggestions have been made to place the burst buffers in the compute nodes as well [93]. In literature, Ni et al. [76] and Bautista Gomez et al. [43] uses SSDs to relieve memory pressure on DRAM. Several supercomputers that will be built between 2016-2019 such as Cori, Summit, and Aurora, all plan to include persistent memory in each compute node in the form of an SSD [13].

Our system uses **application-level checkpointing** in which the programmer carefully selects the data to be saved such that the program can be successfully restarted with that data. The data is written out in the format of a file, and storing and retrieving the file is handled by the file system on the SSD. Usually, when writing a file to any storage device, it is first temporarily allocated in the memory then flushed to the device later. To ensure the file has persisted to the SSD, the Linux *fsync()* operation must be called after each checkpoint. Otherwise, there can be no guarantee the file is recoverable after a crash and reboot.

4.3.3 Checkpoint Location Controller (CLC)

The CLC writes checkpoints to the ramdisk or to the SSD by setting the file path to point to either the ramdisk or the SSD. The decision is made just before the application starts writing each checkpoint. The CLC can maximize the lifetime of the SSD (Section 4.3.3.1), and/or minimize the performance loss of the application (Section 4.3.3.2). It can also take into account the size of the checkpoint (Section 4.3.3.3). An overview is in Figure 4.4. Section 4.3.3.4 shows how all three metrics are combined into one algorithm used by the CLC.

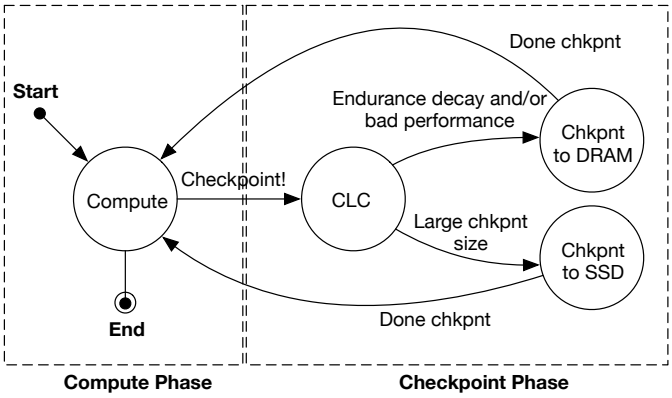


Figure 4.4: This state machine representing application execution shows how in the checkpoint phase the CLC dynamically decides the checkpoint location on each iteration.

4.3.3.1 Lifetime Estimation

The endurance of an SSD is described by bytes written (e.g. TBW–terabytes written or PBW–petabytes written), which is the total amount of writes that it can withstand without wearing out. To obtain an example for the lifetime of a real device, we chose the Intel DC S3700 SSD in 800 GB as a reference [51]. Intel's "DC" data-center SSD's are some of their highest endurance SSDs suitable for high performance computing. The S3700 reported an endurance rating of 14.6 PBW [51].

To measure endurance decay, the CLC calculates an 'expected lifetime' ($L_{expected}$) and an 'estimated lifetime' ($L_{estimated}$). The 'expected lifetime' is a static calculation based on how many petabytes have already been written. For example, a brand new SSD is expected to last 5 years, but as it accumulates writes, the lifetime linearly shortens. The 'estimated lifetime' is a dynamic calculation of how long the SSD might last given the current application's write bandwidth. If the 'estimated lifetime' is smaller than the 'expected lifetime', then that is interpreted as a sign of high usage and accelerated endurance decay. Below are the two equations for this metric.

$$L_{expected} = (PBW_{rating} - PBW_{used}) \times \frac{5\ years}{PBW_{rating}} \tag{4.1}$$

$$L_{estimated} = \frac{PBW_{rating} - PBW_{used}}{B_{SSD}} \tag{4.2}$$

where PBW = petabytes written and B_{SSD} = write bandwidth to the SSD.

4.3.3.2 Performance Loss Estimation

Additionally, the CLC can be configured to monitor the dynamic performance loss of the application as a result of checkpointing to the SSD. If this option is enabled, the CLC monitors the amount of time elapsed since the launch of the program and the fraction of that time spent on checkpointing. We employ a stop-and-copy style checkpointing operation. Just before the next checkpoint, the CLC determines whether the time already lost to checkpointing exceeds the specified bound (e.g. 10%), and if so, directs the next checkpoint to the ramdisk. Each MPI process makes this decision independently.

$$T_{slowdown} = \frac{T_{chk}}{T_{compute} + T_{chk}} \tag{4.3}$$

4.3.3.3 Checkpoint Size

Finally, the CLC considers the size of the checkpoint to determine if there is enough ramdisk space available. Since ramdisk shares the main memory, its size must be limited to avoid swapping from the disk. CLC directs all large checkpoints to the SSD. However, if this decision conflicts with the prior 'lifetime' and 'performance loss' decisions, then the checkpoint is skipped altogether and the application moves on until the next checkpoint interval.

The downside to this approach is that it reduces the number of checkpoints and increases the average rollback distance during recovery. A more severe outcome is unintended uncoordinated checkpointing which can cause the application to restart from the beginning if all the MPI processes cannot agree on single synchronized

checkpoint to roll back to. To avoid such issues, the CLC can potentially be forced to checkpoint on particular intervals.

4.3.3.4 CLC Library

Currently, the controller is written as a library that is added to the application's source code. It can interface with existing application-level checkpointing mechanisms and frameworks such as Scalable Checkpoint/Restart (SCR) [74]. The algorithm used by the controller is provided below. Lines 2-3 call the lifetime estimation and performance loss estimation features and lines 4-11 make a decision based on their results. Lines 12-15 checks the checkpoint size and skips writing large checkpoints to the ramdisk. Lines 16-17 actually writes the checkpoint and updates the checkpoint overhead measurement.

Algorithm 1 Checkpoint Location Controller (CLC)

```
1:  function CLC(D, r, i)        ▷ Where D - data, r - MPI rank, i - chkpnt number
2:      L_estimated, L_expected = lifetimeEstimation()
3:      T_slowdown = performanceEstimation(T_chk, T_total)
4:      if L_estimated > L_expected then
5:          loc = SSD
6:          if T_slowdown > bound then
7:              loc = RAM
8:          end if
9:      else
10:         loc = RAM
11:     end if
12:     size_limit = TMPFS_SIZE/numMPIRanks
13:     if loc == RAM and sizeof(D) > size_limit then
14:         return
15:     end if
16:     writeCheckpoint(loc, D, r, i)
17:     update(T_chk)
18: end function
```

4.3.4 Recovery by Checkpoint Procedure

During restart, the application first searches for a checkpoint file that has been saved by a previous run. An attempt is always made to recover from the checkpoint in ramdisk. If it finds the latest checkpoint in ramdisk, it begins reading in that checkpoint. However, if the ECC logic signals a detectable, but uncorrectable memory error, then the entire ramdisk checkpoint is discarded. Information regarding uncorrectable memory errors can be located by 'edac' ('error detection and correction') kernel modules in Linux. The backup checkpoint file in the SSD is read in if the one in memory was corrupt. The checkpoint in the SSD could be older, leading to a longer rollback distance during recovery. We assume that the SSD has strong ECC built-in that protects its checkpoint and that it is always reliable.

4.4 DRAM ECC Design

The proposed dual-ECC mode memory system has normal ECC for regular data, and strong ECC, that is Chipkill-Correct, for checkpoint data. A typical memory access to a DDR3 x4 memory module containing 18 chips (16 for data and 2 for ECC) reads out a data block of size 512 bits over 8 beats. A Chipkill-Correct scheme can correct errors due to a single chip failure and detect errors due to two chip failures. For x4 DRAM systems, such a scheme is based on a 4-bit symbol code with 32 symbols for data and 4 symbols for ECC parity and provides single symbol correction and double symbol detection. It has to activate two ranks with 18 chips per rank per memory access resulting in high power consumption and poor timing performance [53, 115]. In contrast, the proposed ECC schemes for regular and checkpoint data only activate a single x4 DRAM rank and have strong reliability due to the use of symbol-based codes that have been tailored for this application. Reed-Solomon (RS) codes are symbol based codes that provide strong correction and detection capability

[66]. Here, we propose to use RS codes over Galois Field (2^8) for both normal and strong ECC modes.

Fault Model. When selecting the ECC algorithms for normal and strong ECC, the type of failures and how they manifest in the accessed data are considered. The DRAM error characteristics are well analyzed in [107, 108, 109]. In this work, we assumed errors are introduced by 5 different faults (bit/column/pin/word/chip) [62]. A bit fault leads to a single bit error in a data block. A column failure also leads to a single bit error in a data block. A pin failure results in 8 bit errors and these errors are all located in the same data pin positions. A word failure corrupts 4 consecutive bit errors in a single beat. A whole chip failure leads to 32 bit errors (8 beats with 4 bits/beat) in a 512 bit data block.

Faults can also be classified into small granularity faults (bit/column/pin/word) and large granularity faults (chip). Several studies have shown that small granularity faults occur more frequently than large granularity faults and account for more than 70% among all DRAM faults [107, 108, 109]. Hence, errors due to small granularity faults should be corrected with low latency in any ECC design.

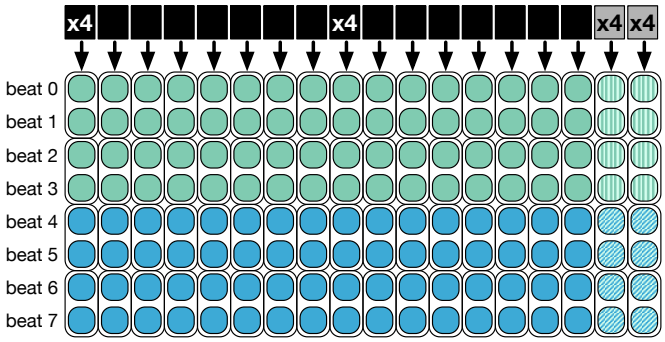


Figure 4.5: The depicted **normal ECC** access reads 512 bits from eighteen x4 chips, two of which are ECC chips. Two beats are paired up to create 1 8-bit symbol per chip. The first 4 and last 4 beats form two RS(36,32) codewords (green and blue).

4.4.1 Normal ECC

Normal ECC provides error correction coverage for regular data accesses, similar to typical ECC DIMMs for servers. It is designed to meet the following requirements:

1. To correct frequent errors due to single-bit/pin/word failures without triggering restart from a checkpoint.
2. To have small decoding latency of syndrome calculation since it is in the critical path of memory access.
3. To activate one rank per memory access and to have better timing/power/energy than Chipkill-Correct.

To satisfy these requirements, we use RS(36,32) over GF(2^8) for normal ECC. It has a storage overhead of 12.5%, which is the golden standard for ECC design [62]. RS(36,32) has a minimum distance of 5 and supports the following setups: (i) double error correction, (ii) four error detection, and (iii) single error correction and triple error detection [66]. If the decoder is designed for setup (i), then 2 symbol errors due to 1 chip failure can be corrected. However, 4 symbol errors due to 2 chip failures cannot be corrected and will lead to silent data corruption [62]. If designed for setup (ii), errors due to 2 chip failures can be detected but small errors due to bit/pin/word failures cannot be corrected. These small granularity faults are reported to occur frequently in memory systems and they must be corrected in order to avoid unnecessary restarts from checkpoints. Setup (iii) can correct all errors due to small granularity (single bit/pin/word) faults in a single chip, detect errors due to 1 chip failure, and has strong detection capability for 2 chip failures. Specifically, for double chip failures, setup (iii) can correctly detect several combinations of two small granularity faults and provide very strong detection for the other cases. Based on this reasoning, RS(36,32) with setup (iii) is chosen to protect normal data.

Results will later show that the normal ECC scheme has a very low silent data

corruption rate of 0.003% and a small latency of 0.48ns for the syndrome calculation. Furthermore, since only 1 rank is activated in each memory access, it has better timing/power/energy performance than the traditional x4 Chipkill-Correct scheme.

Memory access pattern: As illustrated in Figure 4.5, upon a memory read, one rank with 18 chips are activated and 512 bits are read out over 8 beats. Each beat contains 4 bits from a single chip, thus two beats can be paired to form an 8-bit symbol in an RS codeword. The $18 \times 2 = 36$ symbols from the first 4 beats are sent to one RS(36,32) decoding unit followed by the second set of 36 symbols from the next 4 beats. If a codeword has 1 symbol error, it is corrected and sent to the last level cache (LLC). If an uncorrectable error (i.e. ¿1 erroneous symbol) is encountered, then a flag is set. In such a case, the OS would see the flag, terminate the application, and trigger rollback and restart from the checkpoint. Upon a memory write, the ECC encoder forms two RS(36,32) codewords and stores them in a DRAM rank as in a normal memory write.

4.4.2 Strong ECC

Checkpoints that are stored in DRAM memory have to be protected by a strong ECC mechanism to preserve the integrity of the checkpoint data. The proposed strong ECC is designed to meet two requirements:

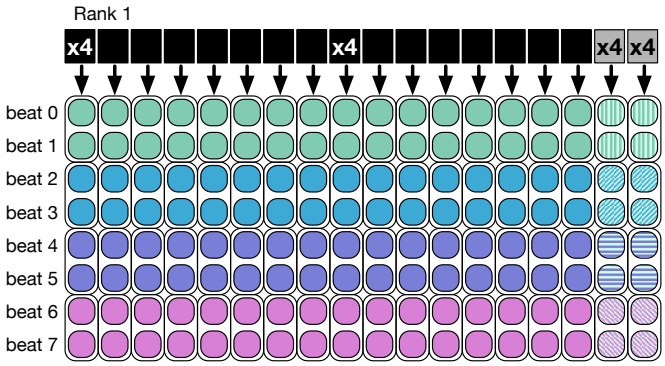
1. To provide Chipkill-Correct level reliability, which can correct all errors due to a single chip failure and detect all errors due to two chip failures. The strong error correction capability reduces the probability of accessing the SSD's checkpoint during restart.
2. To require minimal differences in hardware so as to be able to switch easily from and to normal ECC. Since ramdisk pages can be mapped anywhere in physical memory, the DRAM modules should be flexible in holding normal or checkpoint data without special modifications to the DRAM devices.

We propose using RS(19,16) over GF(2^8) for strong ECC. It works by a hierarchical two-layer scheme where 18 out of the 19 symbols are stored in one rank and the 19th symbol (the third parity symbol) is stored in another rank, as in V-ECC [122].

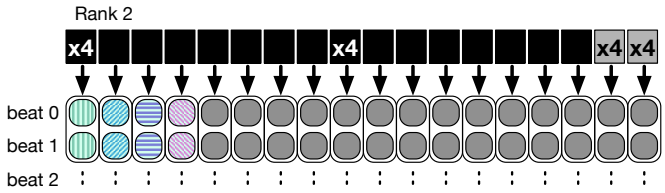
The two-layer scheme works because of the embedded structure of the RS code [66]. The parity check matrix of RS(18,16) is embedded in the parity check matrix of RS(19,16) and thus these two codes can share the same decoding circuitry. The two symbols in the syndrome vector of RS(18,16) are identical to the first two symbols in the syndrome vector of RS(19,16). Once RS(18,16) detects errors, the third ECC symbol can be used to generate the third symbol of the syndrome vector of RS(19,16) and then the RS(19,16) decoder can perform error correction [66].

A direct implementation of this scheme would result in two memory accesses thereby degrading performance and incurring higher power consumption. Thus, an ECC cache is employed to store the third parity symbol and hide the latency due to the extra read and write accesses as in [122]. Additionally, activating just one rank per memory access has better timing/power/energy compared to conventional Chipkill-Correct.

Memory access pattern: As illustrated in Figure 4.6a, upon a memory read, only one rank is activated and 18 symbols (16 data + 2 ECC) are sent to the RS(18,16) decoder. Every two beats of data form one RS(18,16) codeword. The RS(18,16) decoder is designed to perform detection only. Note that RS(18,16) can detect up to 2 symbol errors (2 chip failures). If it detects errors, the decoder is halted and the third parity symbol is fetched from the ECC cache and sent to the RS(19,16) decoder. If the ECC cache does not have the parity symbol, then a second memory access is used to get it from another rank (Figure 4.6b). RS(19,16) can perform single symbol correction and double symbol detection (SSC-DSD) and can thus provide Chipkill-Correct level protection. If the RS(19,16) decoder detects an uncorrectable error, then the entire DRAM checkpoint is discarded and the application retrieves a potentially



(a) Strong ECC, Memory access 1



(b) Strong ECC, Memory access 2

Figure 4.6: (a) Strong ECC creates four RS(18,16) codewords (green, blue, purple, and pink); each codeword is based on 2 beats of data; (b) If errors are detected, four additional ECC symbols are retrieved to form four RS(19,16) codewords.

older checkpoint from the SSD. The recovery procedure was outlined in Section 4.3.4.

Upon a memory write, 512 data bits are encoded into 4 codewords. Two of the parity symbols in each codeword are stored in the two ECC chips in the same rank by a regular memory write operation. The third parity symbol is stored in the ECC cache or in another DRAM rank.

4.4.3 Modification to the Memory Controller

The strong ECC mode exists simultaneously with normal ECC that protects regular memory data; and only requires modification to the memory controller, not the DRAM devices. In order to identify ramdisk/checkpoint data, the page table can be marked with a special flag to indicate ramdisk pages. As illustrated in Figure 4.7, regular data is routed via the normal encoder/decoder and ramdisk data is routed via the strong encoder/decoder. We rely on an ECC address translation unit to determine

the location of the second memory access for strong ECC as in V-ECC [114].

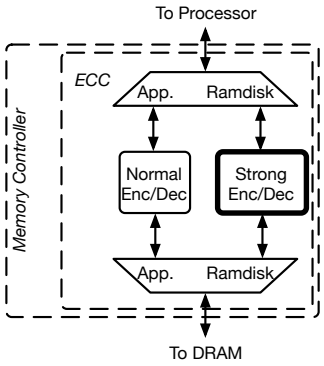


Figure 4.7: Modified Memory Controller with two decoders for normal and strong ECC.

4.5 Evaluation Setup

4.5.1 Microbenchmark

A microbenchmark was written to evaluate the performance of writing a wide variety of checkpoint sizes to different platforms. It was written as an MPI program in C++ to simulate typical parallel supercomputing applications. It mainly consists of two phases: compute and checkpoint. The compute phase runs an algorithm which takes roughly 5 seconds to finish, and the checkpoint phase writes a file of a specified size to either the ramdisk or the SSD. The microbenchmark runs for 100 total iterations of the two phases.

The microbenchmark can be launched with any desired number of MPI processes. To take our measurements, we ran the microbenchmark with 64 MPI processes across 8 nodes. The desired checkpoint size is passed into the microbenchmark as an input, and the same size of checkpoint is made in all 100 iterations. Although there are some supercomputing applications whose checkpoint sizes vary during runtime, most applications save a particular data structure such as the <x,y,z> position vectors of particles or a vector of temperatures. Thus, having a fixed checkpoint size throughout

is acceptable.

We measured the total runtime of the microbenchmark under three naïve implementations i) no checkpointing, ii) checkpointing to ramdisk only, and iii) checkpointing to SSD only. The results, which were already shown in Figure 4.1 in Section 4.2, indicated that writing the checkpoint to ramdisk incurs only a small slowdown of 14%, whereas the SSD incurs a 4.6× slowdown.

4.5.2 MPI Barrier Synchronization Latency

In a globally coordinated checkpointing approach, the overall checkpointing latency consists of two parts: coordination time and storage access time. In the coordination part, all the MPI processes across all the nodes are synchronized via a global MPI barrier. In our simulations, although we did not explicitly measure this latency, it is captured in the total runtime to completion in our simulations.

With many improvements to the global interconnection network, the coordination latency to synchronize the MPI barrier is relatively fast even in large networks. A 750-node Cray XC system using an Aries interconnect reported the end-to-end latency (from source compute node to destination compute node) of an 8-byte MPI message to be $1.3\mu s$ [15]. On the other hand, previous studies have found that the storage access latency is the dominating part of the overall checkpointing latency, as much as 95% [37, 40]. There have been proposals to group MPI processes into many small groups and synchronize only the MPI processes within that group rather than a large global synchronization [41, 47]. These advances further reduces the coordination latency.

4.5.2.1 Typical Checkpoint Sizes

Checkpoint sizes can be reported for an MPI process, for a node, or for an entire application. It is difficult to determine real checkpoint sizes unless real HPC applica-

tions are run at scale on a supercomputer. Even though mini-apps and proxy-apps are representative of the algorithms of the HPC applications, one of their shortcomings is that they are not representative of the runtime or the memory size of large HPC applications.

We conducted a survey of past literature to determine typical checkpoint sizes. An older version of NAS Parallel Benchmark suite checkpointed 3.2MB-54MB per process [49]. MCRENGINE, a checkpoint data aggregation engine, was evaluated on applications having checkpoint sizes between 0.2MB-154MB per process [52]. An experiment on Sierra and Zin clusters at LLNL wrote 50MB and 128MB per process, respectively [85]. A PFS-level checkpointing evaluation on two large clusters HERMIT and LiMa wrote 294MB and 340MB per process, respectively [104]. Note that often times more than one MPI process runs on a multi-core node. Node level checkpoint sizes have been reported between 460MB-4GB/node [76].

To illustrate the wide variety of existing checkpoint sizes, our microbenchmark experiments use between 100MB-1000MB per MPI process; and we run 8 MPI processes per node.

4.5.3 Proxy-apps

The proposed Checkpoint Location Controller was validated against two real benchmarks: *miniFE* and *Lulesh*. *miniFE* is a proxy-app whose main computation is solving a sparse linear system using a conjugate-gradient (CG) algorithm. In a checkpoint, miniFE saves solution and residual vectors. *Lulesh* is a proxy-app that models shock hydrodynamics. It solves a Sedov blast problem while iterating over time steps. In a checkpoint, Lulesh saves the vectors for energy, pressure, viscosity, volume, speed, nodal coordinates, and nodal velocities. The simulation setup and the parameters used to run the benchmarks are given in Table 4.2. The parameters were decided upon using instructions that came with each application on how to scale up

the problem size given the available memory in each node, which was 24GB in our servers.

Table 4.2: Simulations parameters for *miniFE* and *Lulesh*

	miniFE	Lulesh
Parameters	528×512×768	45×45×45
Setup	64 MPI processes, 8 nodes 24GB/node	
Checkpoint sizes:		
1 MPI proc:	50 MB	8 MB
1 node:	400 MB	64 MB
App. Total:	3.1 GB	512 MB
Baseline runtime:	236 sec.	74,470 sec.
Checkpoint behavior:	once/iteration, ∼1 sec/iter, 200 iterations,	once/iteration, ∼11 sec/iter, 6,499 iterations

4.5.4 SSD Device Reference

We chose the Intel DC S3700 SSD in 800 GB using a SATA 3 6Gbps connection for our experiments [51]. It reported an endurance rating of 14.6 PBW and a maximum sequential write speed of 460 MB/s. We were able to achieve write speeds of only 250 MB/s during our checkpoint experiments. The write bandwidth to the SSD is important because faster writes lead to less application slowdown and less overall power consumption. There is a PCIe version of the same SSD available with higher bandwidth; however PCIe is more expensive. On CDW-G, a popular IT products website, Intel's PCIe-based SSDs for data centers retail at upwards of 92¢ per gigabyte. On the other hand, their SATA SSDs retail as low as 71¢ per gigabyte.

4.6 Results

4.6.1 Controller Results

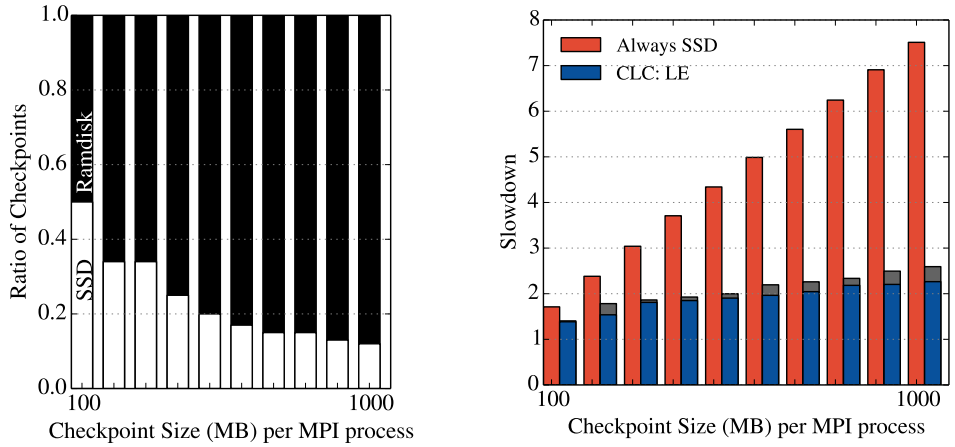
4.6.1.1 Lifetime Estimation

The first set of results are with only the lifetime estimation (abbreviated **LE**) feature. Again, the controller uses Eq. 4.1 and Eq. 4.2 (Section 4.3.3.1) before each checkpoint to determine if the current rate of checkpointing by the application will prematurely wear out the SSD. We assumed an endurance rating of 14.6 PBW (on a brand new SSD) that leads to 5 years of useful life.

Each node has a local SSD and the controller takes into account the endurance of the local SSD and the cumulative bandwidth of 8 MPI processes in the node writing checkpoint files to it. As can be seen in Figure 4.8a, once the endurance is taken into account, fewer checkpoints are written to the SSD, especially at larger checkpoint sizes. At 1000MB per process, only 12% of checkpoints are written to the SSD. Advantageously, this leads to a performance improvement; the slowdown of the benchmark is considerably lessened to an average of only 1.9× (Figure 4.8b)—as opposed to the nearly 8× slowdown (4.6× on average) if always checkpointing to the SSD.

The shaded region above each bar for the CLC's results in Figure 4.8b indicates the overhead due to encoding the checkpoint data with strong ECC before writing to the DRAM. In experiments, the overhead of a second memory access was simulated by writing the checkpoint twice to DRAM. Using this method to measure ECC overhead predicted about 20% additional slowdown, making the average slowdown about 2.1×. This is a worst case estimation of the ECC overhead; in practice, the second memory access can be optimized by using an ECC cache for parity symbols of strong ECC.

Figure 4.9 shows the improvement in endurance gained by the endurance-aware checkpoint controller. This result was obtained after the application completed, and

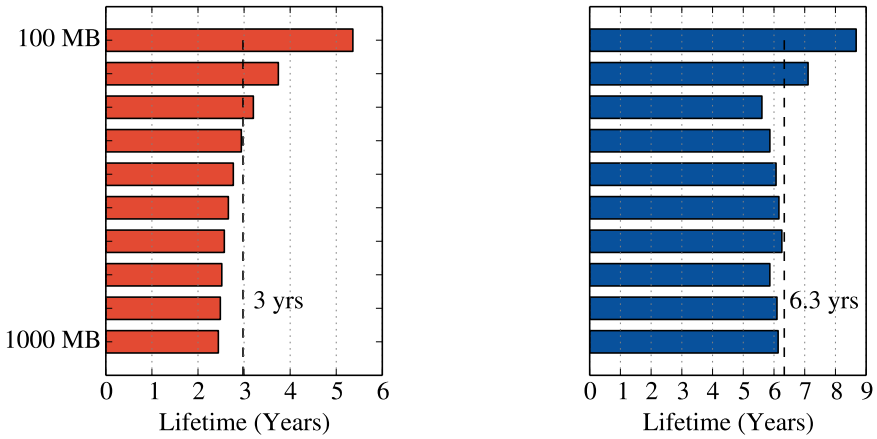


(a) Checkpoint Location

(b) Slowdown

Figure 4.8: Microbenchmark results with the CLC's lifetime estimation (LE) feature enabled. (a) For bigger checkpoint sizes, more checkpoints are written to the ramdisk. (b) The CLC significantly reduced the slowdown. The shaded region above each bar is the overhead for strong ECC's second memory access.

was based on its runtime and how many checkpoints it wrote to the SSD. If checkpoints were only written to the SSD as in Figure 4.9a, then the SSD is estimated to last an average of 3 years across all the checkpoint sizes. On the other hand, the LE feature of the controller extended the SSD lifetime to an average of 6.3 years (Figure 4.9b), ensuring that users can get the guaranteed 5 years of life from their SSD.



(a) SSD Only

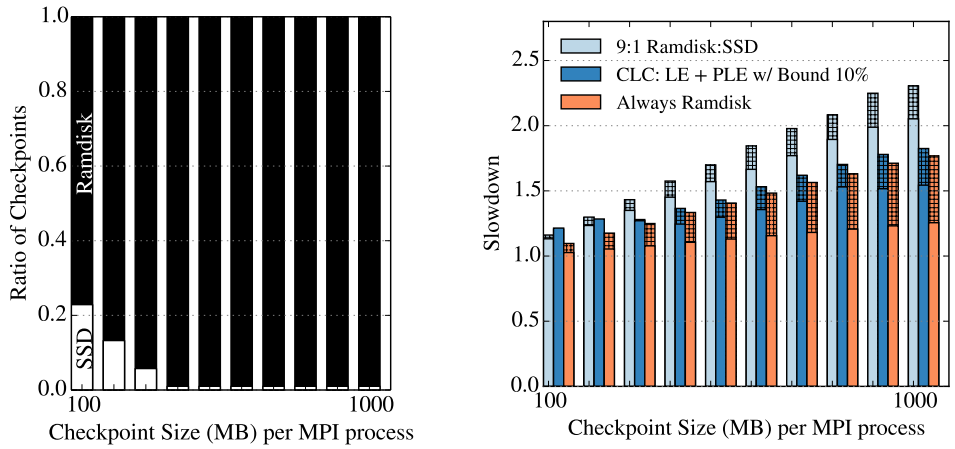
(b) Controller: LE

Figure 4.9: Expected lifetime of the SSD is improved with the LE feature in the CLC.

4.6.1.2 Performance Estimation

Although, the controller was able to successfully prolong SSD endurance, the application still experienced 2.1× slowdown, as was shown in Figure 4.8b. To further minimize performance loss, with the LE feature still enabled, we also enabled the performance loss estimation (abbreviated **PLE**) feature. The performance loss bound was set to 10% in this experiment. Note that the 10% bound was optimistic because even the 'always-ramdisk' checkpoint experienced 3%-25% slowdown across the checkpoint sizes.

As Figure 4.10a shows, the controller wrote even fewer checkpoints to SSD when the PLE feature was enabled; almost 99% of checkpoints were written to ramdisk. Nevertheless, it was successful in decreasing slowdown even further to only 36% on average (47% with strong ECC overhead). More importantly, the controller's achieved performance is more closer to the 'always-ramdisk' approach which achieved 14% slowdown on average (42% with strong ECC overhead).



(a) Checkpoint Location

(b) Slowdown

Figure 4.10: (a) Performance loss estimation (PLE) feature attempts to contain the performance loss within a specified bound (e.g. 10%) and leads to even fewer checkpoints to the SSD. (b) PLE's improved slowdown is closer to ramdisk's. Shaded regions above each bar represent worst-case overheads from strong ECC encoding.

For the sake of comparison, we also implemented a naïve scheme where every 10th

checkpoint is written to the SSD, labeled as "9:1 Ramdisk:SSD" in Figure 4.10b. This scheme performed better than CLC's smarter scheme for checkpoint sizes of 100 MB and 200 MB per process, indicating that a fixed scheme might be sufficient for applications with small checkpoint sizes that want to achieve a balance between performance and reliability. However, across all checkpointed sizes, it's average slowdown was 58% (72% with strong ECC overhead), that is 22% worse than CLC's PLE feature. The ratio 9:1 was arbitrarily picked; a larger ratio can be chosen for even smaller performance loss if the DRAM checkpoint has strong ECC protection.

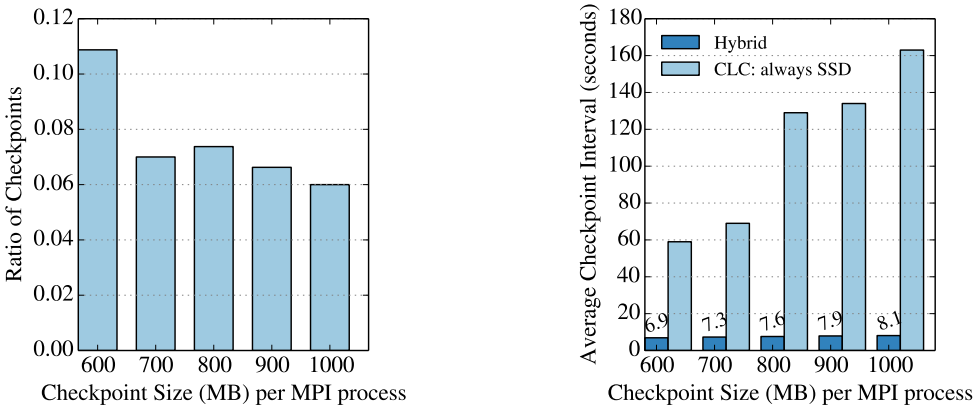
4.6.1.3 Checkpoint Size

CLC's size checking feature is configured to direct checkpoints bigger than a particular size (e.g 0.5GB) to the SSD, that is, these large checkpoints are never written to the ramdisk. In this configuration, some checkpoints maybe skipped if the LE and PLE features indicate unfavorable results. Figure 4.11a shows that for checkpoint sizes 600MB and bigger, the CLC wrote less than 10% of the intended number of checkpoints. It also increased the average checkpoint interval of this microbenchmark (ideally a 5-second interval) from less than 10 seconds to 1-2.5 minutes (Figure 4.11b).

Skipping checkpoints leads to longer rollback distances and, more severely, to unintended uncoordinated checkpointing (Section 4.3.3.3). To avoid such issues, the CLC can be changed forcefully write particular checkpoints.

4.6.1.4 Energy

Energy saved from writing checkpoints to the DRAM is an additional benefit of our proposed hybrid method. First, we measured the power consumed during a checkpoint operation to both the ramdisk and the SSD. Power measurements were obtained via the "watts up?" meter and its smallest sampling rate is 1 second. It measures the load of one entire server node; thus, the measured power includes



(a) Checkpoints Made to SSD

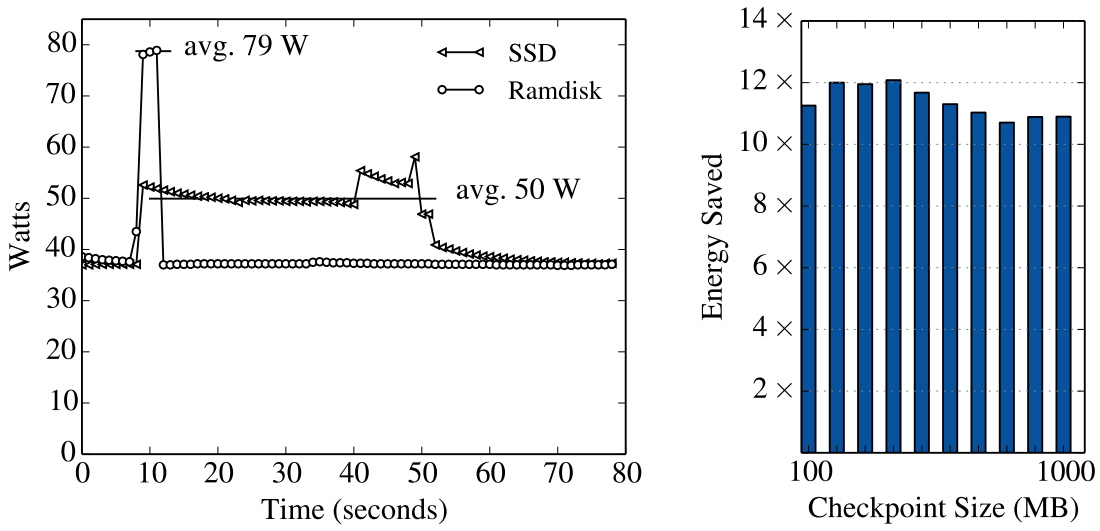
(b) Avg. Checkpoint Interval

Figure 4.11: (a) With CLC's size checking feature, big checkpoints are always written to the SSD. But this leads to only a small fraction of checkpoints actually being written, while the rest are skipped. (b) This feature drastically increases the average checkpoint interval.

everything from CPU, DRAM, I/O bus, SSD, and more.

Figure 4.12a shows the node's power consumption while continuously writing a 10GB file. We chose a very large file size to obtain a measurable power sample because writing small files to the DRAM is very fast (under 1 second) and does not get picked up by the "watts up?" meter. The idle power of the server was 37W and checkpointing to the SSD saw a jump to 50W on average. Interestingly, checkpointing to DRAM registers much higher power consumption at 79W on average. However, writing to the DRAM took only 3 seconds compared to the 42 seconds for the SSD. Overall, DRAM uses less energy because of its speed advantage.

Second, the power numbers obtained from the power profile and the ratio of checkpoints sent to the ramdisk vs. SSD were used to calculate the total energy consumption during checkpointing. Figure 4.12b shows that between 10×-12× energy savings were gained from the checkpoints that were written to the ramdisk instead of the SSD. These results demonstrate the energy savings with only the LE feature from Figure 4.8.



(a) Power Profile

(b) Energy Savings

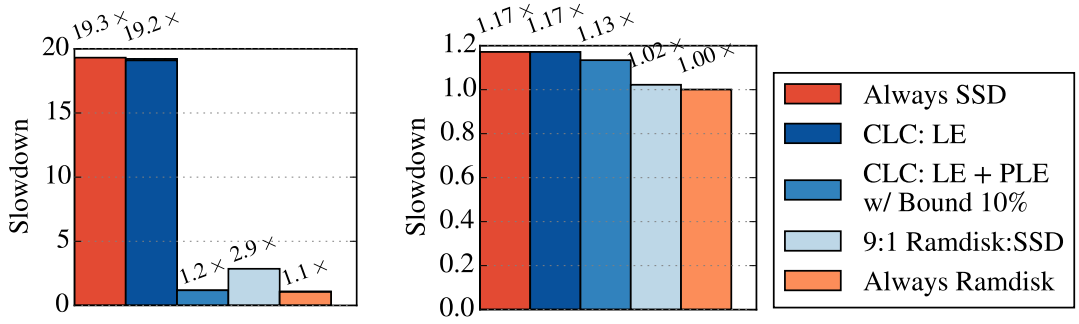
Figure 4.12: (a) The SSD consumes 50W during a write operation, whereas the DRAM consumes 79W. (b) However, due to DRAM's faster write bandwidth, redirecting some checkpoints to the DRAM saves overall checkpoint energy.

4.6.1.5 Real Applications

Finally, we evaluated the CLC with real benchmarks *miniFE* and *Lulesh*. *miniFE* wrote 50MB checkpoints per MPI process with only 1 second of computation in a checkpoint interval. At a node level, 8 MPI processes write 400MB of checkpoints each iteration. As can be seen in Figure 4.13a, the 'always-SSD' approach caused nearly a 19× slowdown, as did the CLC with LE feature enabled. The slowdown is a consequence of the frequent checkpoint behavior of this application. However the checkpoints were small enough not to cause premature wearing out of the SSD; hence, the CLC directed almost all checkpoints to the SSD. Enabling the PLE feature with a bound of 10% was able to decrease the slowdown to 1.2×, but then the CLC directed almost all checkpoints to the ramdisk. In comparison the "9:1" scheme that sent 1 out of 10 checkpoints to the SSD saw a 2.9× slowdown and 'always-ramdisk' approach saw a 1.1× slowdown.

Figure 4.13b shows the results for *Lulesh*, which wrote very small 8MB checkpoints per MPI process at a sufficiently large interval of 11 seconds. Since the bandwidth

to the SSD is low enough so as not to cause accelerated endurance decay, the CLC always chose the SSD. Enabling the PLE feature reduced the performance loss from 17% down to 13% by redirecting 17% of all checkpoints to the ramdisk. With only 2% slowdown, Lulesh is an example of an application that might be better suited for a static "9:1" scheme that balances out both reliability and performance.



(a) miniFE

(b) Lulesh

Figure 4.13: Neither miniFE nor Lulesh checkpoints with high enough bandwidth to wear down the SSD; thus CLC's LE feature allows most checkpoints to the SSD. Enabling the PLE feature, on the other hand, makes the CLC re-direct most of miniFE's checkpoints to the DRAM.

4.6.2 ECC Overhead & Error Coverage Results

The performance overhead of ECC on application runtime were already included in results in Figures 4.8-4.13. This section focuses on synthesis and error coverage results for ECC.

4.6.2.1 Synthesis

We synthesized the decoding units of RS(36,32), RS(18,16) and RS(19,16) codes over GF(2^8) using 28nm industry library. The syndrome calculation is performed for every read and so we optimize it for very low latency. The decoding latency of syndrome calculation is 0.48ns for RS(36,32) code and 0.41ns for RS(18,16) and

RS(19,16) codes. Thus the syndrome calculation latency is less than one memory cycle (1.25ns if the DRAM frequency is 800MHz).

For normal ECC, if syndrome vector is not a zero vector, RS(36,32) performs single symbol correction and triple symbol detection. It takes an additional 0.47ns to correct a single symbol error or declare that there are more errors. For strong ECC, RS(18,16) is configured to only perform detection. If the syndrome vector is not a zero vector, the memory controller reads the third ECC symbol and forms the RS(19,16) code. After calculating the syndrome vector for RS(19,16), the decoder spends an additional 0.47ns to correct a single symbol error and if it cannot correct the error, it declares that there are more errors. The synthesis results are shown in Table 4.3.

Table 4.3: Synthesis results for proposed RS codes

	RS(36,32)	RS(18,16)	RS(19,16)
Syndrome Calculation	0.48ns (σ)	0.41ns (ρ)	0.41ns (ρ)
Single Symbol Correction & Double Symbol Detection	N/A	N/A	ρ + 0.47ns
Single Symbol Correction & Triple Symbol Detection	σ + 0.47ns	N/A	N/A

4.6.2.2 Error Coverage

The reliability of four ECC schemes, namely, RS(36,32) for normal ECC, RS(18,16) and RS(19,16) for strong ECC, and x4 Chipkill-Correct was evaluated. 10 million Monte Carlo simulations for single bit, pin, word, and chip failure events were conducted. Each fault type was injected into a single chip or two chips. For each type of error event, the corresponding detectable and correctable error (DCE) rate, detectable but uncorrectable error (DUE) rate and silent data corruption (SDC) rate [62] were calculated; Table 4.4 gives the corresponding simulation results for these four ECC codes.

RS(36,32) for normal ECC can correct all errors due to small granularity faults

Table 4.4: The error protection capability

Failure Mode	RS(36,32)	RS(18,16)	RS(19,16)	Chipkill-Correct
Single Chip Failures				
1 bit	DCE: 100% DUE: 0% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%
1 pin	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%
1 word	DCE: 100% DUE: 0% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%
1 chip	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%
Double Chip Failures				
1 bit + 1 bit	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 bit + 1 pin	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 bit + 1 word	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 bit + 1 chip	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 pin + 1 word	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 pin + 1 pin	DCE: 0% DUE: 99.9999% SDC: 0.0001%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 pin + 1 chip	DCE: 0% DUE: 99.9969% SDC: 0.0031%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 word + 1 word	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 word + 1 chip	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 chip + 1 chip	DCE: 0% DUE: 99.9969% SDC: 0.0031%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%

and can detect all errors due to a single chip failure. For faults across 2 chips, it can fully detect errors due to a single bit fault in each chip, a single bit fault in one chip and a single pin fault in another chip, and several other error events as shown in Table 4.4. This code has good detection capability for errors due to a pin fault in each chip, 1 pin fault in one chip and 1 chip failure and double chip failures.

The combination of RS(18,16) and RS(19,16) that is used for strong ECC achieves Chipkill-Correct reliability. Recall that RS(18,16) is activated every time to provide detection. It can detect all errors due to double chip failures, and once errors are

detected, RS(19,16) decoder is activated. It can correct all errors due to a single chip failure and detect errors due to double chip failures and thus it achieves Chipkill-Correct level reliability.

4.7 Related Work

Zheng et. al [124] proposed to pair two processors in a *buddy* system where each process makes two identical checkpoints to its own local storage and to the buddy's local storage. The default local storage is the local memory, known as *double in-memory checkpointing*; if a local disk is available then *double in-disk checkpointing* can be carried out instead. At recovery, one of the two buddies provides the restoration checkpoint. Similar to our results, their in-memory checkpoint was faster, but the disk was more practical for applications with big memory footprints. We believe that our two methods can be combined to form a better hybrid-buddy checkpointing method where instead of wasting memory by storing double checkpoints to attain resilience, either our ramdisk or SSD checkpoint can be stored at the buddy's node.

Rajachandrasekar et. al [84] proposed a new in-memory file system called CRUISE (**C**heckpoint **R**estart in **U**ser **S**pac**E**) in which large checkpoints to main memory can transparently spill over to SSD storage. CRUISE is mounted similarly to a ramdisk. Our work can augment CRUISE by providing the necessary strong ECC protection for memory checkpoints. Similarly, CRUISE's spill feature can augment our CLC for checkpoints that are too large to fit in memory. CLC's lifetime estimation feature can provide CRUISE with important information about the endurance of the SSD/spill device.

Saito et al. [89] investigated improving energy consumption during checkpoint write operations to a PCIe-attached NAND-flash device. They suggest that there exists an optimal number of I/O processes that can simultaneously write to the device. They minimize energy consumption by applying DVFS and keeping an I/O profile

that helps to quickly determine the optimal number of I/O processes. This work could possibly be added to our CLC as a new "energy estimation" feature and help predict energy consumption for an energy-limited system that checkpoints to SSDs.

Yoon and Erez [114] proposed Virtual ECC (V-ECC) to protect memory systems with strong ECC mechanisms without modifying existing DRAM packages. This idea makes it possible to provide large parity even for systems that have no dedicated parity hardware. We borrow their technique to provide strong ECC protection for our checkpoints, where the extra parity symbols for strong ECC is stored like data.

4.8 Summary

Exascale supercomputers have millions of components that can fail. A 100 petabyte memory system—100× larger than ORNL Titan supercomputer's 1 petabyte memory system—alone consists of millions of DDR4 DRAM devices backed by hundreds of thousands of SSD flash devices. Resilience to failing components must be achieved by creating a fast and reliable checkpoint/restart framework.

In this chapter, we proposed a hybrid DRAM-SSD checkpointing solution to achieve speed and reliability for local checkpointing while also reducing the endurance decay of SSDs. The Checkpoint Location Controller (CLC) that we implemented monitors SSD endurance, performance degradation, and checkpoint size to dynamically determine the best checkpoint location. CLC running on a microbenchmark showed an SSD lifetime improvement from 3 years to 6.3 years. Application results on *miniFE* and *Lulesh* validated that the online controller can make appropriate decisions to limit the slowdown due to checkpointing.

Furthermore, our normal ECC provides low-latency correction for errors due to bit/pin/column/word faults and our strong ECC provides Chipkill-Correct capability to DRAM checkpoints to reduce the overheads of rollback. The system presented in this chapter demonstrates that it is in fact possible to build an exascale memory

system using commodity DRAM and SSD and gain both speed and reliability without relying on emerging memory technologies.

CHAPTER V

Improvements to Checkpointing with a DIMM-based SSD

5.1 Introduction

Local checkpointing to storage inside the compute node has been proposed to overcome long checkpointing latencies to the parallel file system (PFS) [18, 33, 74, 92]. There are a couple of ways of doing I/O to local storage. One is to use synchronization primitives (`fsync()`, `O_SYNC`) to fully persist the data to the underlying storage. This is usually slow because the I/O function will block until all the data is written at the SSD's programming speed. It is often used in stop-and-copy checkpointing for simplicity. The other is to use non-blocking I/O in which the function returns almost immediately and the data is copied to the SSD at the operating system's discretion. While fast, the non-blocking method provides no persistence guarantees.

One solution to this problem is to use a background thread to perform I/O while the foreground thread continues processing. However, the problem with this method is that a separate process will cause resource contention for cores and caches. Furthermore, spawning a background process with `fork()` triggers Linux copy-on-write semantics where the background process duplicates memory pages that the foreground process tries to modify. The in-memory duplication increases memory bandwidth and

occupies additional memory space.

In this chapter, we proposed a method of doing *partially non-blocking I/O* that also provides persistence guarantees to the application. Although the previous hybrid framework provided persistence guarantees, it did so using an `fysnc()` synchronization call that was fully blocking. The proposed method enables the application to notify an I/O controller about which memory regions need to be saved and resume processing without blocking for the entire copy operation. The application blocks only when it attempts to modify a memory region that has not yet finished checkpointing. One of our main design goals was to minimize additional main memory footprint and main memory bandwidth used by checkpointing; therefore, unlike the copy-on-write method, the proposed method waits for the checkpoint to finish rather than creating in-memory copies.

To engineer the proposed I/O method, we used newer DIMM-based SSDs. In contrast to conventional SATA/PCIe-attached SSDs that require traversing the I/O hub chipset, DIMM-based SSDs place flash storage on the memory bus and offers a tightly coupled connection between DRAM main memory and storage. This type of SSD allows small, cache line-sized transfers to be made from main memory to storage rather than large, block-sized transfers typically made by DMA engines to SATA/PCIe-based SSDs. Furthermore, the SSD Controller (or I/O controller) can make I/O requests directly to the DDR memory controller on the shared memory bus, which is faster than the handshaking protocols usually employed by DMA engines over the PCIe bus. In addition, the shared memory controller's ability to see both which memory regions are being modified and which memory regions have already been saved to the SSD allows us to easily give persistence guarantees to non-blocking I/O.

At the same time, we proposed two optimizations to further hide checkpointing latency to the SSD. These optimizations are aided by the partially non-blocking I/O to the DIMM-based SSD mentioned above. The first optimization *condenses and*

consolidates many sparsely updated DRAM pages into a few flash pages. Condensing and consolidating was designed to amortize the microseconds-long programming latency of flash over as much data as possible. Condensing is enabled by tracking dirty cache lines in every physical page; then the SSD Controller requests only those dirty cache lines for copying. Cache lines across many physical pages are consolidated into one flash page assuring that the checkpoint size is reduced. Note that condensing and consolidating would help even blocking I/O and also works for other flash-based systems such as SATA/PCIe-attached SSDs.

The second optimization *overlaps checkpointing with application execution by beginning copying pages earlier than the start of the checkpoint phase and continuing later than the end of the checkpoint phase.* This optimization is ultimately a further improvement to checkpointing via non-blocking I/O. *Early checkpointing* minimizes the amount of blocking on unfinished checkpointing regions later on. *Late checkpointing* takes advantage of the fact that some pages are infrequently modified and checkpoints them lazily. Both *early* and *late* checkpointing exploits cold periods in updates to memory pages.

Our proposed design strives to reduce the time overhead of checkpointing to the SSD. As compared to the conventional stop-and-copy method, our consolidate method improved average performance by 36% on simulations performed on SPEC CPU2006 benchmarks. In the worst case, where densely updated pages cannot be condensed and consolidated, the proposed design will not be worse than the stop-and-copy performance. The overlapping method improved average performance by 33% over stop-and-copy method. Applied together, they improved average performance by 73% over stop-and-copy method.

In summary, we made the following contributions:

- **Fast checkpointing to SSD.** First and foremost, we proposed methods to hide local checkpointing latency to the SSD over conventional stop-and-copy

without incurring significant slowdowns or additional memory footprint.

- **Partially non-blocking I/O.** We proposed to use DIMM-based SSDs to perform partially non-blocking I/O. The application sets up the SSD Controller to save a region of its memory space and resume processing. Subsequently, the SSD Controller copies the data by making small, cache line-sized requests to the DDR memory controller. This method allows I/O latency to be hidden without losing the persistence guarantee.

- **Consolidation.** We proposed to condense and consolidate sparse updates to main memory pages into a few flash pages in order to amortize the microseconds-long program latency of flash over as much data as possible.

- **Early-Late Checkpointing.** We proposed to utilize cold periods in memory pages to checkpoint them earlier or later than the intended checkpoint phase. The early-late method both benefits from and further improves the use of partially non-blocking I/O for checkpointing.

5.2 Motivation and Background

5.2.1 Checkpointing to Conventional SSDs

It is desirable to obtain a guarantee of persistence after the checkpoint has been made to the SSD to protect against a future failure that may corrupt or wipe out the volatile main memory and caches. According to the Linux `man` pages, obtaining that guarantee requires using either `fsync()` or `O_SYNC`/`O_DSYNC` flags with `write()`. Forced synchronization such as these are often performed as blocking I/O where the application stalls and waits until the I/O function returns after completing the entire copy operation. In the case of writing to the SSD, this incurs long latencies attributed to slow SSD programming speeds.

Alternatively, file I/O can be overlapped with application execution using a separate background thread or process. But, employing a background process for each foreground MPI process creates contention for CPU resources in a shared many-core environment. Also, as mentioned earlier, the background process invokes copy-on-write semantics that consumes additional bandwidth for in-memory duplication.

Furthermore, the copy operation itself is performed by a DMA engine. After the application initializes a `write()`, and after the filesystem location is resolved, the device driver allocates a DMA buffer, places the data there and hands over the handle to the DMA engine. Placing the data in the DMA-accessible region of memory is a privilege of the kernel, which makes it difficult for an application employing non-blocking I/O to check whether that data has been persisted. In Linux, raw I/O (`O_DIRECT` flag) directly from user space to the SSD almost always must be synchronous, that is, the `write()` system call cannot return until the operation is complete [88].

Finally, the DMA operation itself involves handshaking protocols to access main memory. When the DMA engine is granted access to main memory, CPU requests are suspended until the DMA engine releases its hold on the memory bus. DMA setup costs are better amortized over large transfer sizes, but large transfers may block CPU requests for too long. In another mode known as "cycle stealing mode", both CPU requests and I/O requests alternate on the memory bus, but the handshaking process to setup the DMA becomes costly.

5.2.2 New Opportunities with DIMM-based SSDs

Whereas SATA, SAS, PCIe, and NVMe all connected to the processor package via an I/O hub and PCIe switches, DIMM-based SSDs will connect via the memory bus. Higher memory bus bandwidths of 12.8GB/s (DDR3-1600) or 19.2GB/s (DDR4-2400) also means that storage is no longer bottle-necked by the I/O links. While some

I/O latency is reduced, the DIMM-based SSD storage is still accessed by invoking the filesystem, block layer, and device driver code for easier adoption into existing systems. Accessing memory, on the other hand, requires no kernel intervention. While all the ways of doing I/O (blocking, background process, and direct) with conventional SSDs are still applicable to DIMM-based SSDs, having storage on the memory bus presents a unique opportunity to offload data transfer responsibilities of the device driver to the hardware.

In the new architecture design that we proposed, the SSD Controller takes responsibility for transferring data between memory and storage and uses the shared memory controller to mediate communication between them. Unlike the DMA engine, the SSD Controller does not perform lengthy handshaking protocols, master the memory bus and block CPU's access to it, or make bulk transfers from memory. Instead, the SSD Controller requests fine-grained cache-line sized memory reads from the memory controller. User space memory can be read directly; therefore, by simply marking those pages as read-only until they are checkpointed automatically guarantees that the application waits until the checkpoint has been persisted before progressing.

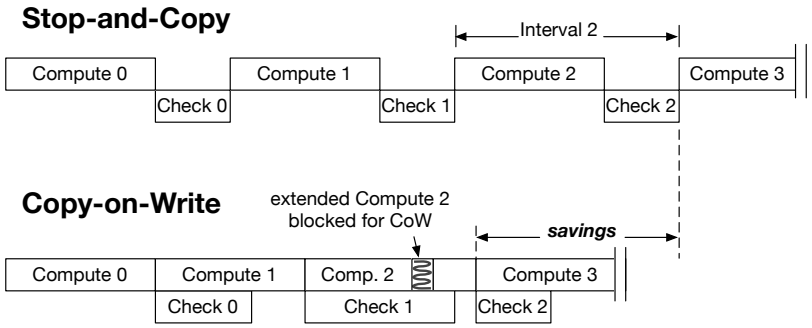


Figure 5.1: Timing overheads by conventional checkpointing methods

5.2.3 Shortfalls of the *Stop-and-Copy* Method

Conventional *stop-and-copy* checkpointing can be applied to DIMM-based SSDs. A software-level approach pauses processing until all data is copied from memory to flash. Alternatively a hardware-level approach in the memory controller stalls write requests. When checkpointing is done, the hardware memory controller resumes write requests. The advantages of *stop-and-copy* is that it is easy to implement and it requires the fewest hardware changes between all of the methods. Its biggest disadvantage is the inevitable stalling and performance loss, as illustrated in Figure 5.1.

5.2.4 Shortfalls of the *Copy-on-Write* Method

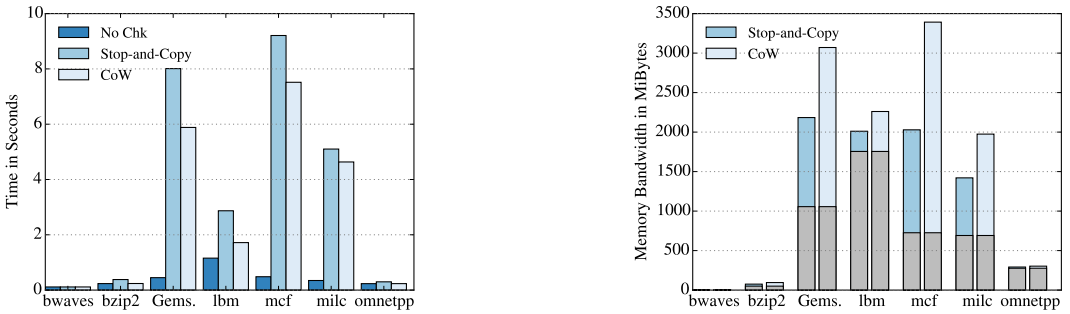
Easily implemented in software, the *copy-on-write* method delegates checkpointing to a background thread while the foreground thread continues execution. The operating system marks all pages "read-only" in the process's page table until updated pages are copied from memory to storage.

The benefit of *copy-on-write* is that the application's stall time is proportional to DRAM's copying speed, not flash speed. Furthermore, similar to *stop-and-copy*, *copy-on-write* applies after the transition to the next checkpoint interval, thus, it captures and preserves the very last update to each page. Lastly, the duplicate is only created once—upon the very first modification to the page since transitioning to the next checkpoint interval. In Figure 5.1, the beginning of phase `Compute 3` is reached far soon than the *stop-and-copy* method.

The biggest downside to *copy-on-write*, however, is that it requires additional DRAM main memory space. By default, the Linux `fork()` operation requires the background thread to duplicate memory pages that the foreground thread wants to modify. In the worst case in which the application wants to modify all memory pages shortly after beginning the next interval, the memory footprint will double. Furthermore, DRAM-to-DRAM copying incurs more memory bandwidth on top of

the DRAM-to-flash copying. To manage the memory footprint from spiraling out of control, the system has to prevent more than one duplicate of any page. In order to enforce this policy, uncheckpointed pages more than two intervals old will be forced to finish checkpointing at the cost of stalling. Hence, *copy-on-write* works best if the checkpoint size is small enough such that the copying time to flash is relatively smaller than the checkpoint interval length.

Figure 5.2a and 5.2b are runtime and memory bandwidth results from running 7 of the most memory intensive benchmarks with conventional checkpointing methods. As illustrated, *stop-and-copy* incurs tremendous slowdown and *copy-on-write* incurs a lot of memory bandwidth. *Copy-on-write* performance in Figure 5.2a is almost as bad as *stop-and-copy* for the most memory intensive benchmarks (`GemsFDTD`, `lbm`, `mcf`, and `milc`) because the length of time to copy all modified pages was longer than the checkpoint interval we selected, which forced the application to stall.



(a) (b)

Figure 5.2: Checkpoint results (a) runtime and (b) memory bandwidth with conventional *stop-and-copy* and *copy-on-write* methods. Although *copy-on-write* improves performance, it is at the cost of extra memory bandwidth use.

5.2.5 Using DIMM-based SSDs to Hide Checkpoint Overhead

A key benefit of DIMM-based SSDs is that they are directly accessible via the memory controller, which also manages DRAM main-memory. This direct access in hardware opens up the opportunity to access the file system storage directly without

invoking the kernel. Moreover, it opens up the opportunity to overlap checkpoint data movement with application execution without the use of a background thread/process. A large capacity DIMM-based SSD with smart wear-leveling in the FTL should maintain the same endurance guarantees as an equivalently-sized SATA/PCIe SSD. Moreover, the shorter data path over the memory bus reduces the energy consumption of the checkpoint over a conventional SSD.

To guide our search for a better checkpointing methodology that functions with the DIMM-based SSD setup, we defined the following three design principles:

1. Optimizations proposed to hide the overhead of writing the checkpoint to the DIMM-based SSD should not cause slower application performance than the conventional stop-and-copy checkpoint.

2. The checkpointing method should minimize the use of DRAM main memory. Since high density flash storage is now on the memory bus, we should avoid further wasting precious DRAM memory on checkpoints.

3. The checkpointing method should minimize memory bandwidth and memory traffic congestion to running applications. Although memory bandwidth is usually over-provisioned, adhering to this principle saves energy.

We measured the above three principles in our experiments with the metrics of application's performance, checkpoint's memory footprint, and overall memory bandwidth, respectively. Before we delve into the proposed methods, we give a brief background on DIMM-based SSDs and NVDIMMs in the next subsection.

5.2.6 A Brief Background into Flash on the Memory Bus

Bringing flash memory onto the DRAM memory bus interface can eliminate between 5% and 25% percent of I/O access latency associated with kernel functions, PCIe/SATA interface protocols, and data transfer across links [120]. High density

flash devices can expand memory capacity at low cost, opening up opportunities to run memory-intensive applications unlike ever before. Although there are some research ideas [42, 86] and industry ventures into using emerging non-volatile memories (ReRAM, PCRAM, and STT-MRAM), such as 3D-XPoint DIMMs, flash is still by far the most popular and practical choice of non-volatile memory.

There are two main ways of putting flash on the memory bus: as memory (NVDIMMs) or as storage (DIMM-based SSDs). JEDEC has categorized NVDIMMs into three main types: N, F, and P. NVDIMM-N types have both flash and DRAM on the same DIMM, but, only DRAM is system mapped and accessible by the operating system. Its capacity is equivalent to DRAM (up to 32GB) and it operates at the speed of DRAM. The battery-powered flash is only used for backing up DRAM during system powerdown. NVvault [1] (August 2014) and HybriDIMM [3] (August 2016) by Netlist, SafeStor by SMART Technologies [6], and NVDIMM-Ns by Micron are some of the products that are now on the market. NVDIMM-Ns are not a good choice for checkpointing. Due to the lack of space on the DIMM, NVDIMM-Ns are unable to hide the raw read/program latency using the same techniques that SSDs use such as placing multiple flash devices, applying channel-level parallelism, and installing large DRAM buffers [106]. For example, even though a single flash device offers a bandwidth of only 10-20MB/s [31], one of the fastest PCIe SSDs by Intel offers an impressive 32 channels and a 2.25GB DRAM cache that boosts its sequential read bandwidth to 1.75GB/s and write bandwidth to 1.1GB/s [70]. The limited flash storage on an NVDIMM-N also means that it does not have high endurance and using it as a random access memory will only lower its endurance [39].

NVDIMM-F types and DIMM-based SSDs are all-flash DIMMs. NVDIMM-Fs are memory mapped flash while DIMM-based SSDs are block-oriented filesystems. DIMM-based SSDs strive to be competitors for SATA/PCIe SSDs with capacities as large as 400GB. Although the high bandwidth, low latency memory bus interface

provides fast access to storage as compared to the traditional SSD interface, the kernel still has to be invoked to access the filesystem and block layers. Their endurance will be the same as an equivalently-sized SSD. Diablo Technologies has created two DIMM-based SSD products that they call Memory Channel Storage (MCS): ULLtraDIMM in partnership with SanDisk and eXFlash in partnership with IBM. Diablo Technologies also recently revealed an NVDIMM-F product, Memory1, a fully flash DIMM that is memory mapped [2]. Memory1 expands memory capacity but requires a faster DRAM cache that could act as working memory with hot/cold page migration. Memory1 is neither persistent memory nor persistent storage because it is erased on reboot. Finally, when Intel and Micron releases their 3D-XPoint technology on the DIMM form factor, they will likely be DIMM-based SSDs.

NVDIMM-P types do not exist yet. This is the ultimate hybrid memory where both flash and DRAM are on the same DIMM and both are system mapped. It would combine the persistent memory in the N types along with the block-oriented access in the F types. It would simultaneously offer large capacity and semi-fast access.

5.3 Proposed Work

5.3.1 Partially Non-blocking I/O with DIMM-based SSDs

Although a DIMM-based SSD is similar to a traditional SSD in components and internal structure, one important difference is that the memory controller has purview over both it and the main memory. This shared controller introduces a new opportunity to move data directly between memory and storage without invoking the operating system. Leveraging this unique feature in DIMM-based SSDs, we introduce a new communication protocol between memory and storage that allows *the SSD Controller on the DIMM-based SSD to autonomously request copies of memory pages via the shared memory controller*. Prior to initiating the data movement, the SSD

Controller receives instructions from the operating system's file system and block layer protocols as to the name, location, size of the file, and a list of which memory pages should be retrieved to write that file. With this information in hand, the SSD Controller autonomously sends read requests to the shared host memory controller. Each read request retrieves a cache line-sized data unit until the entire set of memory pages for the complete file has been retrieved. Once the file has been persisted to flash storage, the SSD Controller notifies the kernel of its completion.

To enable this process, the writing of a checkpoint file is initiated the same as before by invoking the kernel and requesting space allocation on the SSD. The kernel's file system code (virtual filesystem, file organization layer, and the flash specific filesystem) determines the appropriate location to write the file, allocates disk space, and retrieves the physical block numbers for the SSD. Rather than continuing on with device driver commands to copy pages from memory to flash, the kernel sends the list of memory locations to copy and the physical blocks numbers to which they should be copied to the SSD Controller. At this point, the application resumes the next compute phase. The division of software and hardware responsibilities are illustrated in Figure 5.3.

A detailed diagram of the copying process is given in Figure 5.4. ❶ The SSD Controller initiates checkpointing by sending the host memory controller a read request. ❷ The read request is placed in the memory's read request queue. ❸ The read request is sent to the DRAM DIMM during its turn according to the scheduling policy. ❹ The response data is placed in the memory's read response return queue, similar to normal memory reads. ❺ When processed, the response is forwarded to SSD write queue. ❻ Finally, when the SSD receives the data it may buffer it or program the flash device.

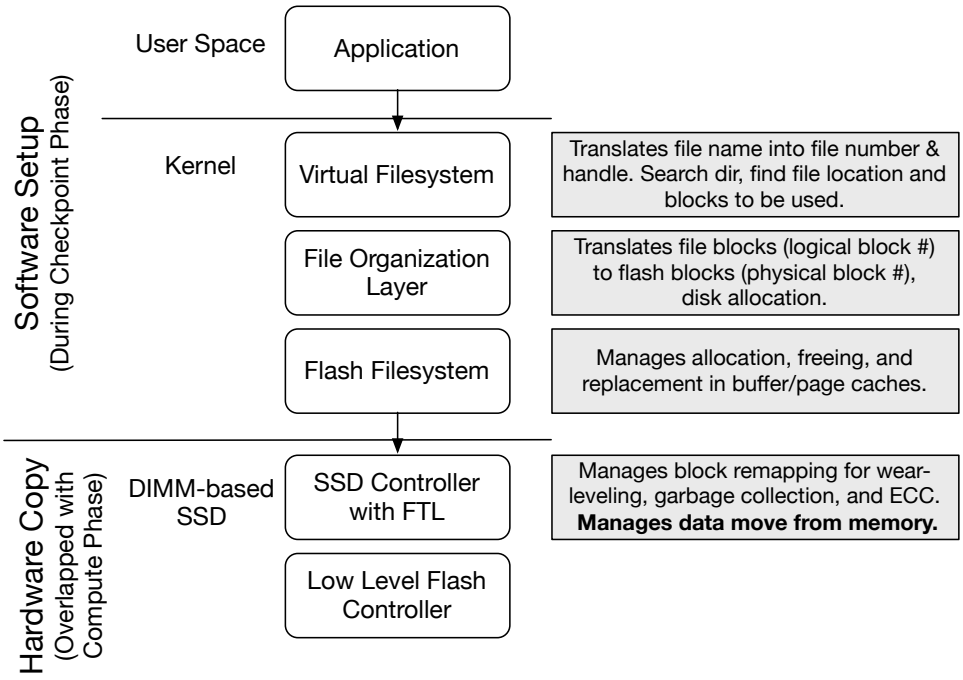


Figure 5.3: Kernel functions are invoked for setting up the checkpoint location in flash storage. Responsibility for data copying from memory to storage is offloaded to the SSD Controller.

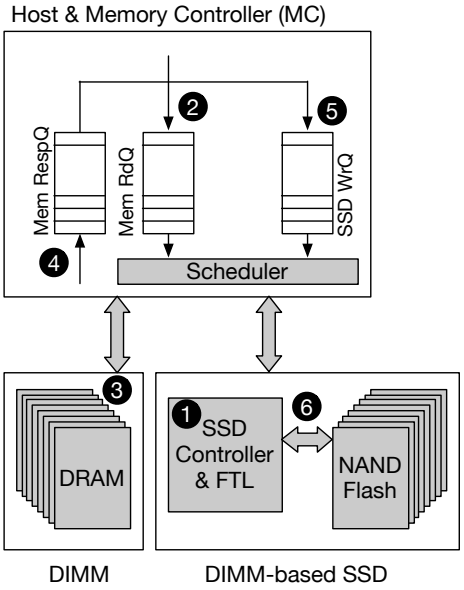


Figure 5.4: Data transfer process from memory to storage. The SSD Controller initiates copying by requesting cache line-sized memory reads from the host's DDR memory controller.

5.3.2 Condensing and Consolidating DRAM Pages

In this chapter, we proposed two optimizations to hide checkpointing latency to the SSD. Our optimizations were guided by our design principles to hide time overhead without using extra memory space or bandwidth. In this subsection, we explain why many of the previously proposed latency-hiding techniques for DRAM and other non-volatile memories (STT-MRAM, ReRAM, and PCRAM) that rely on byte-addressability and low write latency will not work for flash and how the consolidation method is more suitable.

5.3.2.1 Byte-addressable Techniques are Inappropriate for Flash

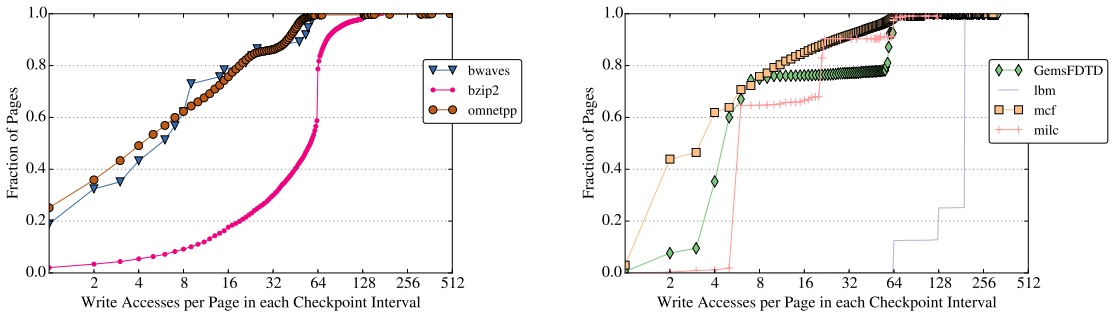
Block remapping in the recent ThyNVM work relied on fast write speeds to use NVM as the "working memory" and remapped a 64-byte cache line to a new memory location upon checkpointing [86]. It is impractical to use flash memory as working memory because it is not byte-addressable and it cannot do in-place updates. First, flash is not byte-addressable because it has to be read and programmed at the granularity of pages. Second, when a cache line is updated, the whole page has to be re-written. An erased flash page starts out with all cells in the logical '1' position and programming the page can only change them to a logical '0.' Therefore, a flash system must always be setup to use DRAM as the working memory with write-backs to flash as checkpointing.

Flash programming time, however, dwarfs DRAM read time. Flash requires at least $200\mu s$ to write a 4KB page compared to the $320ns$ required to read it sequentially from DRAM (assuming DDR4-1600 at $800MHz$). Due to the large disparity in write bandwidths between DRAM and flash, a write-back scheme would require large buffers or queues. In contrast, NVMs that are only marginally slower than DRAM rely on prefetching and the cache hierarchy to hide write-back latencies. ThyNVM hid write-back latency by temporarily applying block remapping within DRAM. That

approach is akin to *copy-on-write* (i.e. page remapping), which, as we discussed earlier, has high memory overhead.

5.3.2.2 Investigating Sparsity of Updates

The constraints of non-byte-addressable flash memory imply that large data transfers are better in order to amortize the long program latency. This observation led us to seek a checkpointing method that combines data and minimizes the number of writes to flash. To this end, we investigated the memory access patterns of a selection of non-memory intensive (Figure 5.5a) and memory intensive (Figure 5.5b) SPEC CPU2006 benchmarks. We collected the number of write requests arriving at the memory controller per page per checkpoint interval. We found that all but two benchmarks had 60% or more pages with fewer than 8 write accesses within one checkpoint interval. About 90% of pages had less than 64 accesses, implying that not all of the 64-byte cache lines in a 4KB page were modified. Only `bzip2` and `lbm` had a majority of entirely updated pages. 23% of `bzip2` pages and 88% of `lbm` pages had more than 64 write accesses. While more than 64 writes do not necessarily imply that they were to unique cache lines, it is a strong possibility that the entire page was updated.



(a) Non-intensive and moderately intensive (b) Memory intensive

Figure 5.5: Number of write requests to a physical page at the memory controller in (a) non-memory intensive and (b) memory intensive benchmarks.

5.3.2.3 Consolidating to Reduce Checkpoint Size

Based on our observations that many pages have sparse updates, we propose a checkpointing method that *condenses sparsely updated physical pages and consolidates them into one flash page*. Our goal is to amortize the program latency of one flash page across as many DRAM pages as possible. Our concept is illustrated in Figure 5.6. In this work, we assumed that each physical page is 4KB, the same size as a typical OS page, and that the minimum modifiable unit is a 64-byte cache line. A single 16KB flash page can hold up to four noncondensed physical pages. Each consolidated entry has 20 bytes of metadata that consists of the physical page address (6 bytes), a bitmap of the stored blocks (8 bytes), and the flash address (6 bytes), which the location of the condensed page in flash. The bitmap serves two purposes: a count of how many 64-byte cache lines are in the entry, and their block offsets. Metadata for each consolidated page is stored separately in a common location for the entire checkpoint.

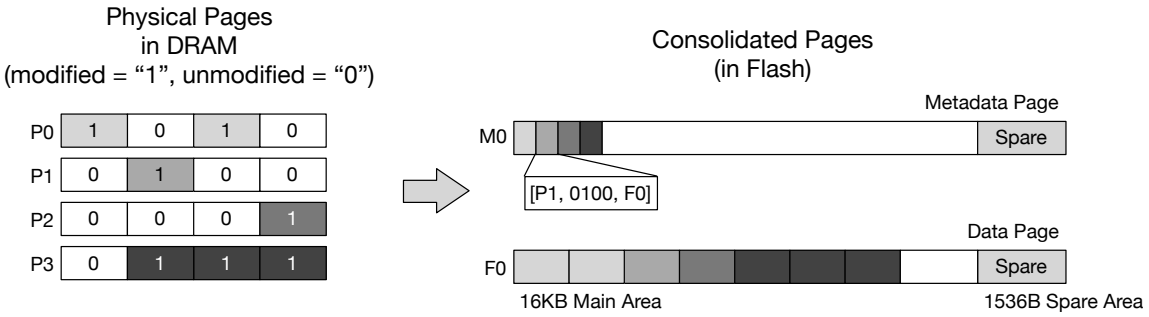


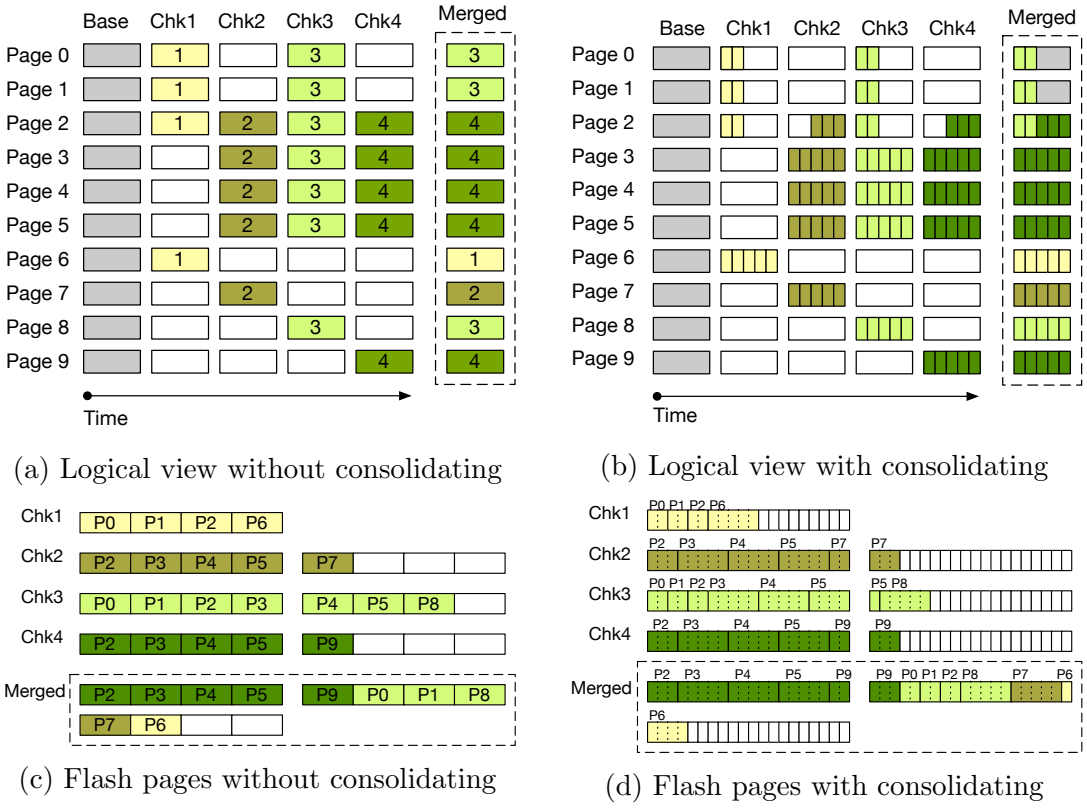
Figure 5.6: Condense and Consolidate Concept

Condensing leverages the concept of sparse granularity of updates. While block remapping leveraged the same concept, condensing applies it across two memory platforms (from DRAM to flash) rather than on the same platform. Besides, unlike block remapping within DRAM, condensing not just hides write-back latency, but decreases it overall without incurring memory overhead.

Consolidation requires only 20 bytes of metadata per 4KB page (0.5% overhead).

In the extreme situation where only one cache line is updated per page, storing 20 bytes of metadata per 64-byte cache line would lead to a 31.25% footprint overhead. Consolidation offsets this overhead by amortizing the programming latency over 256 physical pages (assuming a 16KB flash page). Note that even in the stop-and-copy case, storing noncondensed pages requires 12 bytes of overhead (physical page address and address of its location in flash). Consolidation is similar in concept to compression.

Figure 5.7 presents a detailed comparison of checkpointing without and with consolidation. Figure 5.7a and 5.7c would be how conventional *stop-and-copy* checkpointed. As in *incremental checkpointing*, only updated pages are checkpointed. We omitted showing metadata in this illustration for simplicity.



(a) Logical view without consolidating

(b) Logical view with consolidating

(c) Flash pages without consolidating

(d) Flash pages with consolidating

Figure 5.7: Checkpointing overview (a),(c) without and (b),(d) with consolidating.

Several checkpoints can be merged periodically to gain back free space in flash. We piggyback checkpoint merging on garbage collection. Merging is costly because

it requires reading metadata of all the checkpoints. To minimize the number of write operations, we merge checkpoints backwards in time. For example, in Figure 5.7, merging starts with $Chk4$ and goes backwards to $Chk1$. For consolidated pages, the merging process also reads the bitmaps and figures out which blocks to omit from the merge. For example, *Page 2* in Figure 5.7b contains only the first 2 lines from $Chk3$ and only the last 3 lines from $Chk4$.

5.3.3 Early and Late Checkpointing

Consolidating took advantage of the sparsity of updates, but it cannot hide checkpointing latency when the majority of pages have dense updates. This was true for `lbm` and `bzip2`.

Upon further scrutiny, we noticed that many pages have 'hot' and 'cold' periods. In other words, updates to a page exhibited temporal locality followed by a period of no write accesses. Figure 5.8 shows the write access patterns to pages for all the benchmarks for 5 billion instructions. Checkpoints were taken at intervals of 1 billion instructions (marked as vertical dashed lines).

bwaves (5.8a): The two sloping lines indicate accesses to 2 distinct memory regions. There are no repeating accesses to the same page, thus, `bwaves`'s pages are perfect for overlapping checkpointing with computation.

bzip2 (5.8b): The straight line indicates that the same set of pages are accessed repeatedly. This finding matches Figure 5.5a that showed `bzip2` having many write accesses to a single page. Therefore, `bzip2` is not a great candidate for overlapping.

omnetpp (5.8c): Accesses patterns are mix between `bwaves` and `bzip2`.

leslie3d (5.8d): The first benchmark to demonstrate an iterative pattern. Chunks of pages are modified in short bursts. Lower address regions have longer cold periods and higher address regions have short cold periods.

lbm (5.8e): Walks the entire address space at every iteration. After being touched,

each page has a long cold period for nearly 2 seconds. `lbm` had the highest no. of accesses at 116 million to about 103K pages, verifying the observation made in Figure 5.5b that `lbm` has high locality of accesses per page. `lbm` is a good candidate for overlapping.

milc (5.8f): Pages are walked in a short time. Accesses in low region are spread out across a large number of pages and accesses in the high region are concentrated to a small number of pages. `milc` can benefit only a little bit from overlapping because the iterations are close together and cold periods are not long.

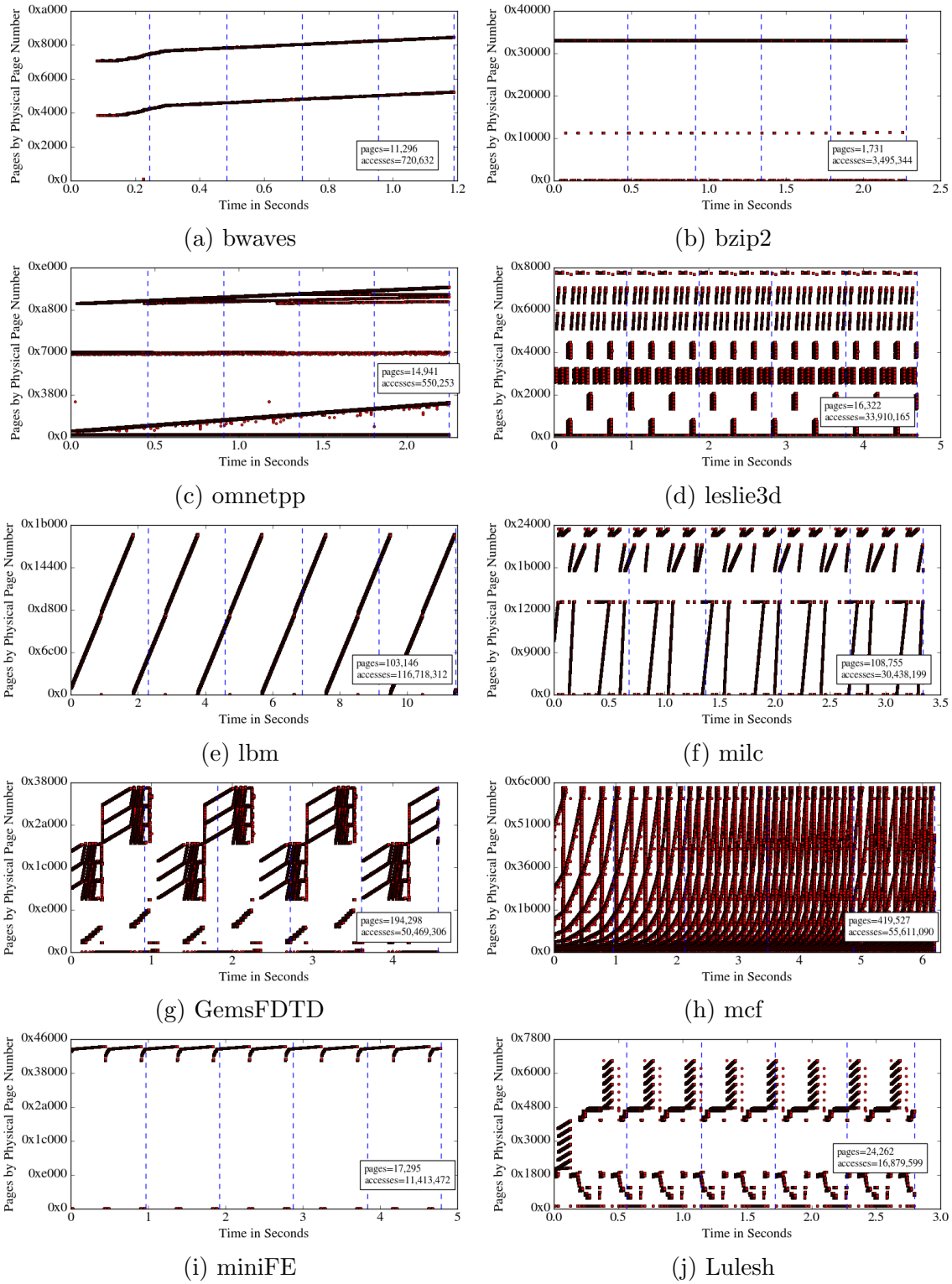
GemsFDTD (5.8g): Accesses seems to be concentrated to 4 distinct regions. Each region is modified as a chunk followed by a long cold period. `GemsFDTD` would benefit from overlapping.

mcf (5.8h): Walks the entire address space in very short iterations. `mcf` is not a good candidate for overlapping.

HPC Apps (Figure 5.8i and 5.8j): Both `miniFE` and `Lulesh` have cold periods, but only of about one-half or one-third of a second. Still, they would benefit from overlapping.

Pages that have very long cold periods in between accesses have more chances to checkpoint without stalling the application. In contrast, trying to checkpoint pages that are frequently updated is likely to stall the application when those pages are locked down. Our investigation revealed that applications with large memory footprints are also more likely to give their pages long cold periods while they are updating another part of memory (e.g. `lbm` and `GemsFDTD`). Our hypothesis is that utilizing these cold periods to overlap checkpointing with application execution will help to hide checkpointing latency.

We split overlapping into two periods called *early* and *late*. In the *early* method, pages are checkpointed during the compute phase before the interval ends. It is good for pages that have a long cold period before the checkpoint phase (before



(a) bwaves (b) bzip2

(c) omnetpp (d) leslie3d

(e) lbm (f) milc

(g) GemsFDTD (h) mcf

(i) miniFE (j) Lulesh

Figure 5.8: Write access patterns to physical pages of SPEC CPU2006 benchmarks for 5 billion instructions. The dashed vertical lines are checkpoint intervals at every 1 billion instructions. The Y-axis shows the address space by physical page number (without the 12-bit page offset). Inside a box in each plot, *pages* indicate the number of unique physical pages and the number of *write accesses* plotted.

the vertical dashed line). In the *late* method, pages that were modified in previous intervals are checkpointed during the computation phases of later intervals. It is good for pages that have long cold period extending well into subsequent compute phases (e.g. `bwaves`).

With regards to *early* checkpointing, predicting when a page enters the cold period would require tracking when the last update to the page arrived, which involves calculating and maintaining a local inter-arrival frequency of updates for each page. Instead, basing off of the observations about write access patterns made in Figure 5.5, we count the number of write accesses to the page to determine whether it is a good candidate for checkpointing early. The write access counter is incremented for every write to the page and cleared at the beginning of the next interval. In the experiments we ran, we waited until a page had 8 write accesses before checkpointing it.

One problem with *early* checkpointing is that in some applications pages have multiple hot periods in the same compute phase. Then, it could be modified again even if it was early-checkpointed. As a solution to miscalculated early checkpoints, we maintain a `modified` bit per each page. On every write access, the `modified` bit is set and the write accesses counter is incremented. If a `modified` page has more than 8 write accesses, then it is a candidate for early checkpointing. When the page is copied to flash, its `modified` bit is cleared. If the `modified` bit is set again within the same interval that means the page has to be re-checkpointed. On a transition to a new checkpoint interval, the write accesses counter is cleared.

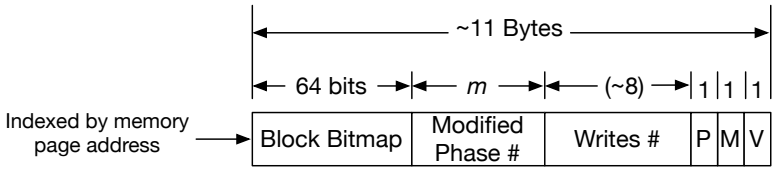


Figure 5.9: A Memory Accesses Tracking Table (MATT). P=this page is top *priority* for checkpointing because a write request is blocked and waiting, M=this page was *modified* again after it was checkpointed early, V=this page is *valid* in memory.

Late checkpointing is a solution to pages that could not be checkpointed early.

The advantage of the late checkpointing method is that the application can transition into the next computation phase without stopping to finish the checkpoint. Our late checkpointing method recognizes that cold periods can cross interval boundaries and that even though there was not enough time during the previous interval, the application does not need to unnecessarily stall before the transition because there maybe plenty of cold time in the next interval before the page is accessed again.

Late checkpointing is a new spin on *copy-on-write* and has one caveat. Unlike the former, which duplicated pages to DRAM, upon a write access to an uncheckpointed page from the previous interval, late checkpointing will stall the application and wait until the page is read out to flash. This approach may create performance loss and may face adverse situations if the application touches all pages shortly before and after a checkpoint interval transition. However, by avoiding duplicating pages in DRAM, we abide by our 2^{nd} principle to minimize DRAM usage. In order to minimize performance loss, we make any uncheckpointed pages that are stalled on the critical path of application execution a top-priority for the next round of flash programming.

Another key idea that makes *late checkpointing* a success is that we allow checkpoints to span multiple intervals (as opposed to conventional overlapping methods where current pages must be checkpointed by the end of the following interval.) In order to implement this, each page records its last modified phase #. When the page is checkpointed even at a much later interval, the SSD Controller knows to which interval's checkpoint it should append the current page. For example, interval 1's pages can begin checkpointing early in interval 1 and continue into interval 4. Drawing from the illustration in Figure 5.7, *Page 6* could still be uncheckpointed in interval 4. A page cannot be modified across intervals, however, until its previous changes are checkpointed.

For maximum performance gains we combine early and late checkpointing into a single method which we call *early-late checkpointing*. The early-late method con-

verts the notoriously bursty I/O behavior of checkpointing into non-bursty I/O by spreading out the flash bandwidth utilization over time.

Finally, we illustrate the timing overheads of the newly proposed methods by a diagram similar to the one that was in Section 5.2. Figure 5.10 shows that condensing and consolidating shorten the checkpointing time and improves the stop-and-copy approach. Early and late, on the other hand, is an overlapping method that is different from stop-and-copy. It completely hides the checkpointing overhead other than when it has to block during an update to an uncheckpointed page.

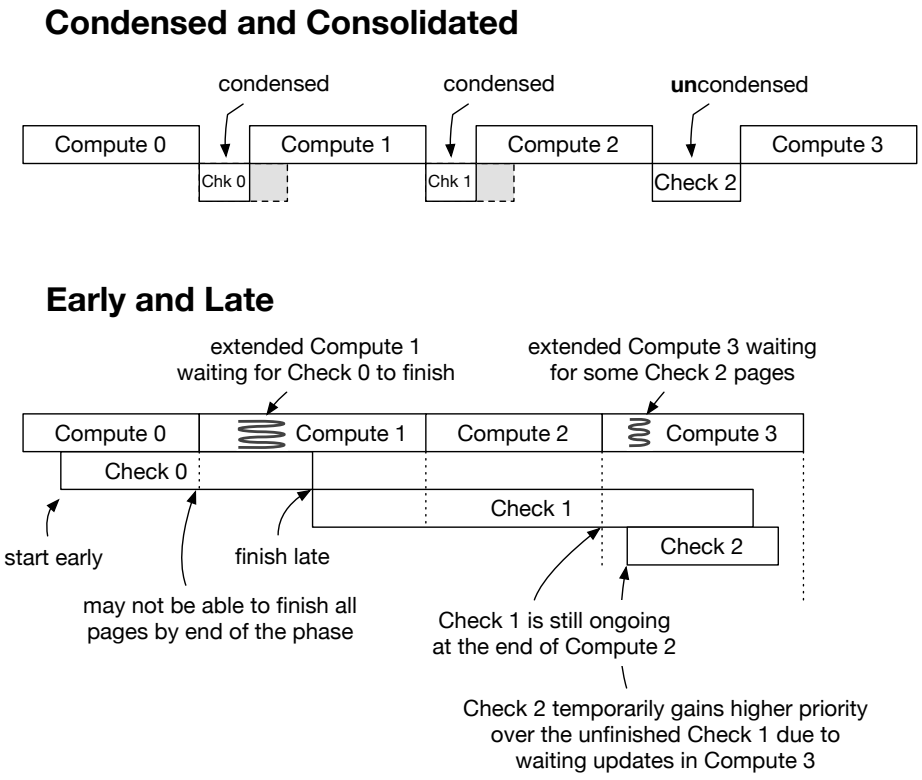


Figure 5.10: Timing overheads by the new checkpointing methods

5.3.4 Area Overhead

We introduce a new Memory Accesses Tracking Table (MATT) to help with partially non-blocking I/O. It is indexed by a page's physical address and tracks the modification to each physical memory page. Figure 5.9 shows a MATT entry, which

contains a bitmap of the page's modified cache blocks (64 bits), the phase in which it was last modified (a variable m bits), the number of writes accesses to it (8 bits is sufficient, since we only want to differentiate between sparse and dense writes), a priority bit, a modified bit, and a valid bit. Roughly 11 bytes are sufficient per 4KB page. In total, the table would occupy 44MB to track all the pages of a 16GB DIMM.

The host memory controller is modified to receive requests by the SSD Controller. This change makes our design different from existing designs because the SSD Controller will become an active device operating autonomously to conduct data transfers. In contrast, today, DIMMs plugged into the memory bus are passive devices.

5.3.5 Memory Accesses Tracking Table Design

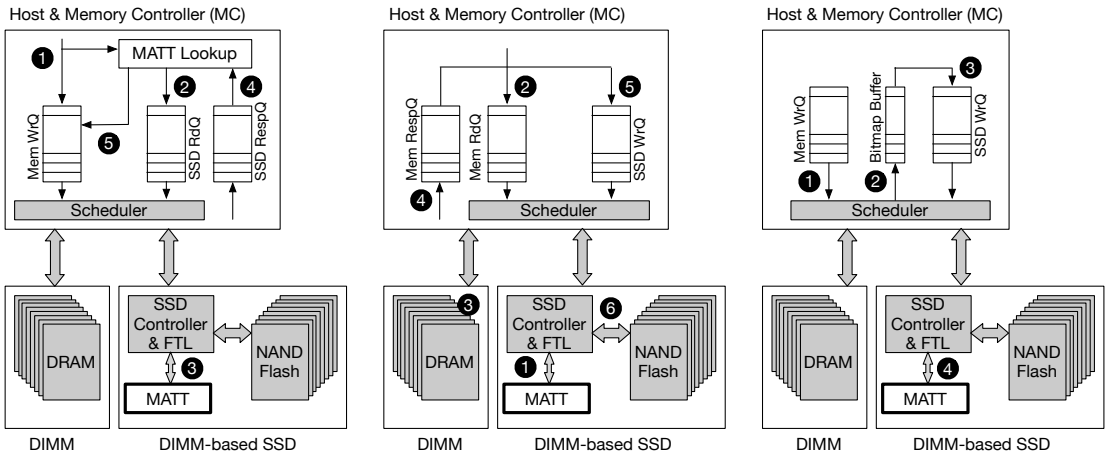
Due to its large size, the MATT is located on the DIMM-based SSD. It is accessed by the SSD Controller on three occasions. On the first occasion, the memory controller looks up the MATT when it receives a write request from the CPU (see Figure 5.11a). Each write request is queued in the Mem_WrQ and the memory controller requests the SSD Controller to lookup the MATT entry to check whether the page belongs to a pending checkpoint or not ❶ - ❸. If the page has not yet been checkpointed, the MATT remembers that there is a waiting write request by marking it a top priority for the next round of flash programming (setting the P bit in the MATT entry). The SSD responds back to the memory controller which marks the request in the Mem_WrQ ready or not ready ❹ - ❺.

On the second occasion, the SSD Controller reads the MATT entries of pages it has to checkpoint (see Figure 5.11b). The block bitmap of each entry identifies modified cache lines. ❷ The SSD Controller initiates checkpointing by requesting the host memory controller to read a dirty cache line. ❸ The memory controller issues read requests to DRAM. ❹ - ❺ Data responses are directly forwarded to the DIMM-based SSD. ❻ Once the flash buffer is full, it is programmed on to the flash device.

If there is a waiting write request, the SSD Controller notifies the memory controller when all the modified blocks of the corresponding page has been copied.

Finally, on the third occasion, the memory controller sends the MATT an updated block bitmap of recently modified pages (see illustration in Figure 5.11c). ❶ - ❷ To avoid generating DIMM-based SSD traffic each time a write request is processed, the memory controller keeps a small associative cache (Bitmap Buffer) of equal size to the write request queue (64 entries) and writes the page addresses and block bitmaps of the write requests it processes. ❸- ❹ Entries are flushed based on an LRU policy and written back to the MATT table on the DIMM-based SSD.

In this design, we assumed that all the dirty cache lines are flushed to main memory prior to starting the checkpoint. In this way, the application is free to modify cached data while it continues processing in the next compute phase. The memory controller can hold off write requests and keep the memory state clean until the checkpoint is finished. If we do not flush dirty cache lines to main memory first and instead tries to write them directly from the caches into the SSD, then that leads to fragmented dirty data and it becomes more difficult to condense a particular page and build a block bitmap that reflects all the dirty lines in both the caches and the main memory. In addition, directly flushing the caches to the SSD is not feasible because the capacity of the combined caches are too big to write very quickly to the flash-based SSD. Finally, the caches have no way to check which cache lines have been checkpointed or not without additional status bits. Therefore, the best approach is to flush the dirty cache lines to main memory at the beginning of each checkpoint.



(a) MATT Lookup (b) Checkpoint Copy (c) MATT Update

Figure 5.11: DIMM-based SSD with a MATT. (a) Write requests to DRAM look up the MATT to ensure that it's not about to overwrite an uncheckpointed page. (b) The SSD Controller scans the MATT and initiates checkpointing. (c) The block number of each write request is saved to the bitmap buffer as it's processed by the memory controller.

5.4 Evaluation Methodology

5.4.1 Simulator and Application Workloads

We used the gem5 simulator in system call emulation mode for our analysis. The gem5 configuration parameters are shown in Table 5.1. The DRAM components in gem5 were used as is. When modeling the checkpointing to the DIMM-based SSD, read requests for copying pages from DRAM to flash are queued in the read queue in the memory controller. The read queue is shared between read requests from the CPU and checkpointing requests from the SSD Controller. Copying an entire 4KB page generates 64 read requests and may cause congestion in the read queues. The flash programming time is modeled as a long delay in between page copying requests.

We evaluate all methods with a selection 8 of memory intensive and non-memory intensive benchmarks from SPEC CPU2006 and 2 HPC proxy apps: miniFE and Lulesh. Table 5.2 presents statistics for each benchmark that indicates their memory intensity. We observed that the commonly used L2 MPKI metric does not best convey

Table 5.1: gem5 simulator configuration.

CPU	1-core, 3GHz, 8-issue, out-of-order
L1 I/D	32KB/32KB, 2-way, private, 64B line
L2	2MB, 8-way, 64B line
Host Memory Controller	Split read/write request queues, and buffering per controller rather than per rank or per bank. Read queue size=32, Write queue size=64. FR-FCFS policy. RoRaBaCoCh. Open-adaptive page policy.
DRAM	2GB DDR3-1600, 1 channel, 8 devices per rank, 1KB page per device tRCD-tCL-tRP = 13.75ns (11 clock cycles)
DIMM-based SSD	SSD Controller has double page buffering. 1 flash channel, 8 packages. 16KB page size, 45μs read, 660μs program, 3.5ms erase. 68MB MATT table.

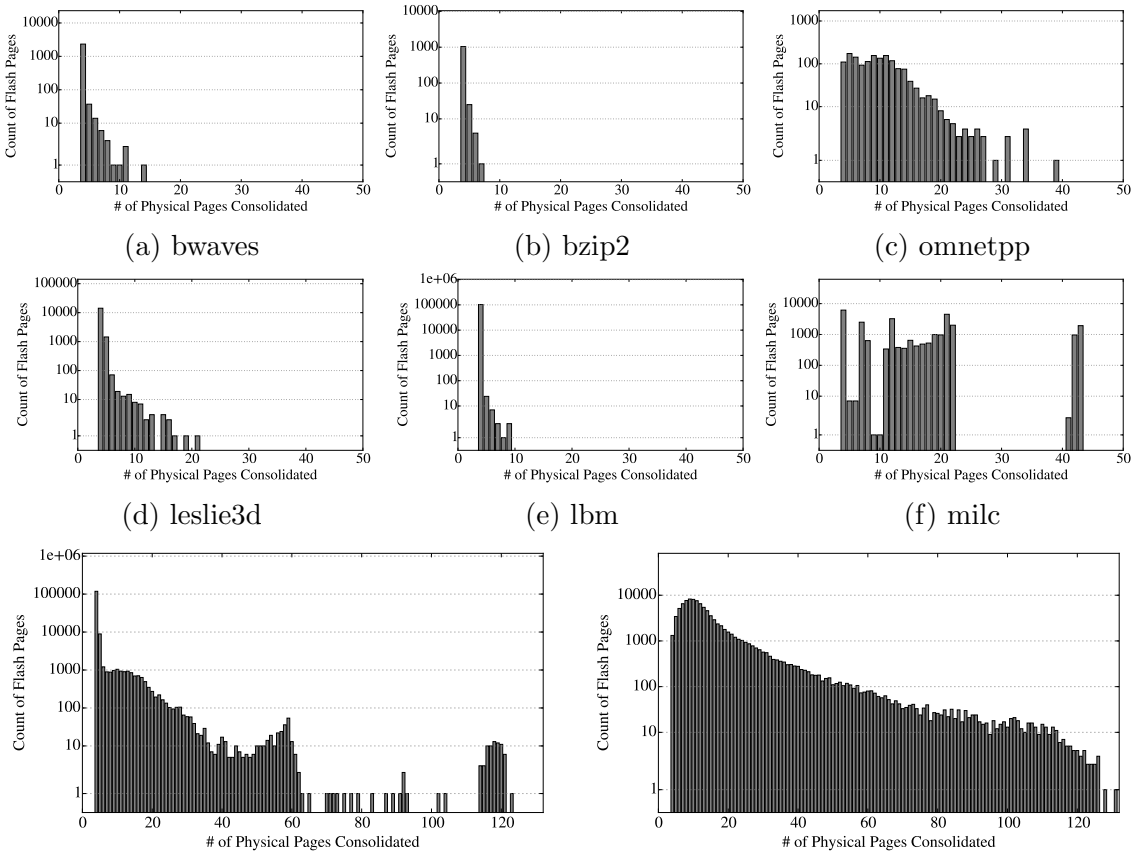
the checkpoint size. For example, `lbm` has the highest MPKI of 31.13 and 116 million write requests to memory. Its average number of modified pages per checkpoint interval, however, is roughly 103K pages, which is less than one-third of the 367K pages that `mcf` modifies with only half as many (55 million) write requests. Therefore we provide the two additional metrics: number of write requests to memory and the average number of modified pages per checkpoint interval.

Table 5.2: Benchmark statistics collected for 5 billion instructions.

	Benchmark	**L2 MPKI**	**Number of Mem. Write Requests**	**Avg. # of Modified Pages / Interval**
Memory Non-Intensive	bwaves	0.17	720,651	2,428
	bzip2	0.98	3,495,375	1,084
	omnetpp	7.94	550,346	3,614
Memory Intensive	GemsFDTD	22.88	50,469,314	179,955
	lbm	31.13	116,718,340	103,146
	leslie3d	21.67	33,910,187	16,322
	mcf	21.30	55,611,218	367,247
	milc	15.97	30,438,219	108,755
HPC Apps	miniFE	37.81	11,413,495	17,295
	Lulesh	7.52	16,879,622	16,857

5.4.2 Checkpointing Setup

Checkpointing in our evaluation is system-level (or hardware-level) where all modified pages are checkpointed as a process image as opposed to application-level checkpointing where the programmer annotates which critical data structures should be to saved to non-volatile memory.



(a) bwaves (b) bzip2 (c) omnetpp

(d) leslie3d (e) lbm (f) milc

(g) GemsFDTD (h) mcf

Figure 5.12: Distribution of flash pages vs. the number of consolidated physical pages they hold. The bigger and longer the tail of the distribution, the more apt the benchmark is for consolidation.

In our simulations, checkpointing for all benchmarks are done at intervals of 1 billion instructions and the benchmarks are simulated for a total of 5 billion instructions. Our chosen checkpoint interval maybe too frequent, especially for applications with large memory footprint. It is up to the system designer to select a checkpoint interval that balances time lost to checkpointing vs. progress lost.

5.5 Results

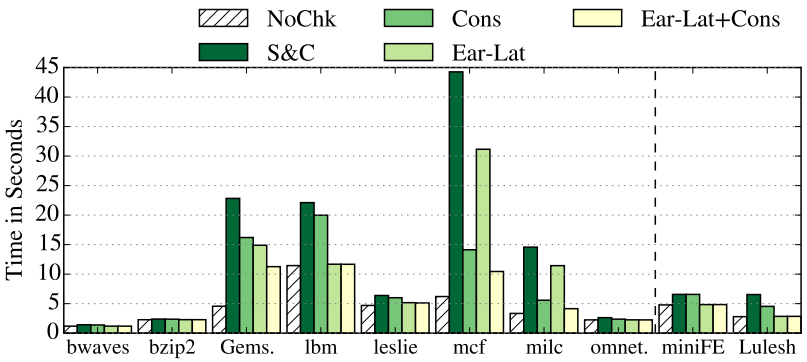
In this section, we present the results for the consolidate method and the early-late method.

Figure 5.12 shows the distribution of flash pages vs. the number of consolidated physical pages per flash page. The minimum number of pages that can be consolidated into a single flash page is 4; because each OS page is 4KB and each flash page is 16KB. As seen, every benchmark had some number of flash pages with more than 4 physical pages consolidated into it. `mcf` and `GemsFDTD`, the 2 benchmarks with the largest memory footprints according to Table 5.2, exhibited long tail distributions, indicating that consolidation was beneficial for them. `GemsFDTD` also did have a significant number of flash pages with only 4 consolidated pages. `lbm` was the worst candidate for consolidation because the vast majority of its pages could not be condensed.

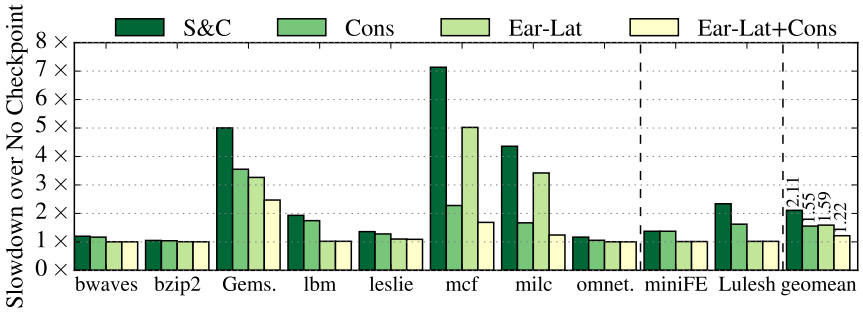
Figure 5.13 shows the runtime results. Figure 5.13a are the simulated number of seconds for 5 billion instructions. Figure 5.13b shows the slowdown over not checkpointing at all. As expected, the stop-and-copy method incured the worst slowdown: 2.1× on average. Consolidation reduced the slowdowns of `GemsFDTD` from 5× to 3.6×, `mcf` from 7.1× to 2.3× and `milc` from 4.4× to 1.7×. Averaged across all the benchmarks, consolidation reduced the slowdown to just 55%. Early-late was the most beneficial to `lbm`, reducing its slowdown from 93% to 2%. It also helped `bwaves` reduce from 20% to 0%, `omnetpp` reduce from 20% to 0%, `miniFE` reduce from 37% to 1%, and `Lulesh` reduce from 62% to 2%. Early-late was not as helpful for `mcf` and `milc`. We mentioned earlier that `bzip2` would not benefit from early-late overlapping, but it reduced the slowdown from 5% to 1%. However, `bzip2` had the smallest memory footprint of all the benchmarks, so its slowdown was not bad to begin with. We reaped the most benefits from applying consolidation on top of early-late overlapping. Together they reduced to the average slowdown to 22%.

Figure 5.13c shows the speedup of the new checkpointing methods over stop-and-copy as the baseline checkpointing method. The results reflect the same observations made in Figure 5.13b.

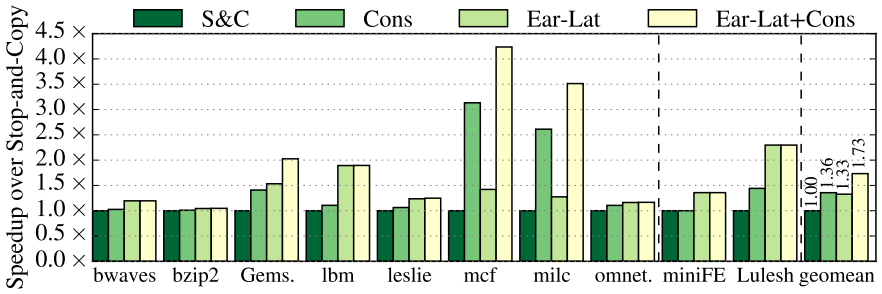
Figure 5.13d shows the overall memory bandwidth. As expected, consolidation



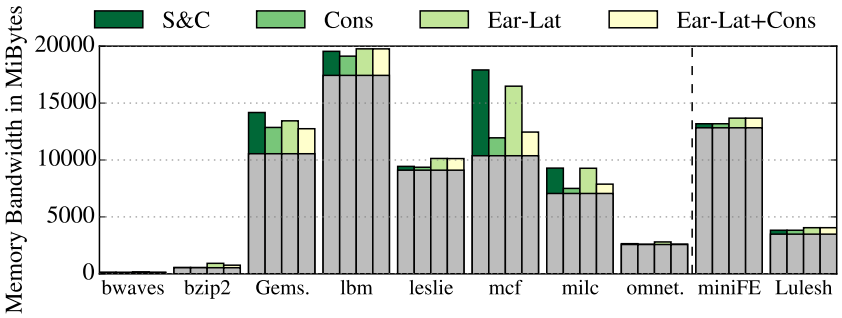
(a) Runtime Seconds



(b) Performance Slowdown over No Checkpoint



(c) Speedup over Stop-and-Copy



(d) Memory Bandwidth

Figure 5.13: Runtime results of the proposed optimized checkpointing methods. Stop-and-copy (S&C) exhibits the worst-case slowdown, consolidate (Cons) is applied on top of stop-and-copy, early-late (Ear-Lat) is the overlapped method.

Table 5.3: Qualitative comparison between conventional and proposed approaches.

Approach	Performance	Memory footprint	Memory bandwidth
No checkpoints	No loss	No extra	No extra
Stop-and-copy	Worst loss	No extra	Checkpointing I/O
Copy-on-write	Better than stop-and-copy if few pages and large intervals	Double in worst case	Checkpointing and memory copy I/O
Consolidate	Better than stop-and-copy if many sparse page	No extra	Equal to or less than stop-and-copy
Early + Late	Better than stop-and-copy if pages have long cold periods	No extra	Equal to more than stop-and-copy

always improves the memory bandwidth. Stop-and-copy used 20% of additional memory bandwidth, averaged across all benchmarks; consolidation reduced this to 11%. Early-late method on the other hand, had worse than expected memory bandwidth (30%), most likely due to multiple hot periods requiring re-checkpointing of early checkpointed pages. Applying consolidation to early-late checkpointing was able to recover the memory bandwidth back to 18%.

Table 5.3 presents a qualitative comparison between conventional checkpointing methods and the proposed optimizations.

5.5.1 Comparison to the Hybrid Framework

In the hybrid DRAM-SSD framework proposed in Chapter IV, we selectively checkpointed to both DRAM and the SSD to balance reliability and speed. Furthermore, checkpointing to the DRAM helped to reduce SSD wearout. Figure 5.14 summarizes scheduling of 10 checkpoints using the hybrid framework. In this section, we will discuss what happens in a system that implements consolidation and early-late checkpointing within a hybrid framework.

First, consolidation could lead to more checkpoints to be written to the SSD. In the example shown in Figure 5.14, consolidation targets just the checkpoints saved to the SSD (namely 0, 3, and 8). Since consolidation makes a single SSD checkpoint faster, more of them can be written to the SSD for the same performance loss tolerance margin. But, saving more checkpoints to the SSD can again lead to more wearout

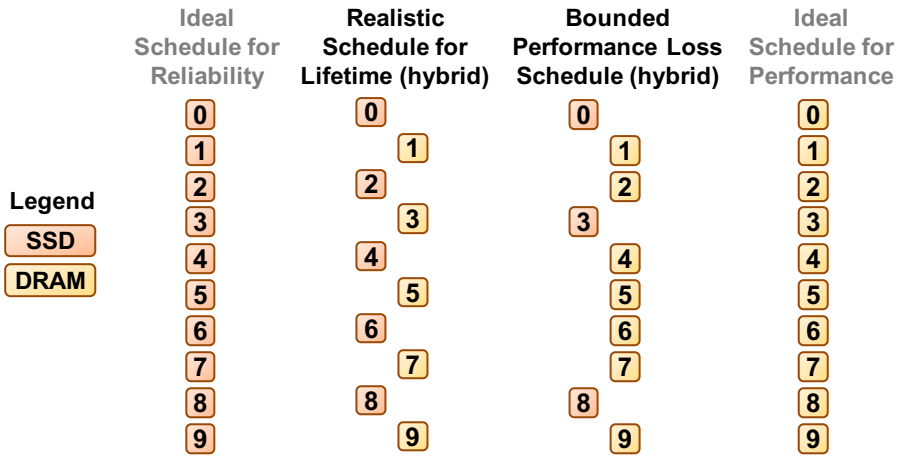


Figure 5.14: Example of checkpointing schedules in the hybrid framework. Assuming the ideal schedule for reliability sacrifices time and assuming the ideal schedule for performance sacrifices reliability. Therefore, a more realistic schedule obtained with the hybrid framework and lifetime estimation minimizes wearout by redirecting every other checkpoint to the DRAM. If the performance loss is still greater than the user set bound (e.g. 10%), the hybrid framework with performance loss estimation redirects more checkpoints to the DRAM.

of the SSD. Therefore, it is important to always use the lifetime estimation feature provided by the hybrid framework to keep the number of checkpoints written to the SSD in check.

Second, the SSD checkpoint can be written with the early-late method without additional changes. Let's examine a situation where the first couple of checkpoints are written to the SSD and the third checkpoint is written to the DRAM. With early checkpointing, writing can begin even before the compute phase ends. Early checkpointing is the same as before. By the end of the first compute phase there has been no visible performance degradation. With late checkpointing, writing continues into the second compute phase. Modifications to the uncheckpointed pages may stall, however, while waiting for late checkpointing to write them to the SSD. By the time the second compute phase ends, the performance degradation due to stalling may have accumulated beyond the user-set maximum performance loss bound. Then, during the transition period from the second to the third compute phases, the CLC

makes the decision that the third checkpoint should be written to the DRAM. Since writing to the DRAM is relatively fast, the third checkpoint does not have to employ the early-late method. Instead, it can be quickly written to the DRAM at the end of the compute phase. Meanwhile, late checkpointing from the first and second phases could be overlapped with the third compute phase and they can continue slowly writing back to the SSD.

To summarize, adopting consolidation and early-late checkpointing within a hybrid framework is the most effective way to hide the performance degradation due to checkpointing.

5.6 Related Work

In this chapter, we studied techniques to reduce checkpoint time in the context of DIMM-based SSDs. Below we summarize the closely related works.

5.6.1 Work on NVDIMMs

DIMM-based SSDs and NVIDMMs are a fairly recent invention and there is limited industry and research documentation regarding it.

Chen et al. [28] explored the challenge of placing an NVDIMM on the memory bus. Specifically, they addressed the issues of mixing I/O and memory traffic on the same channel and the performance degradation caused by it. They proposed to split the transaction queue into two queues in the memory controller in order to address the speed mismatch issue between DRAM and flash. They also proposed a proactive garbage collection design for flash that minimizes data movement. Their work is appropriate for an NVDIMM-F design where communication between the DRAM and the flash has to traverse the memory channel(s) and has to be mediated by the host memory controller. Our work is different from theirs in that we use an NVDIMM-N design. Our I/O requests do not clog up the memory queues because the

SSD Controller only requests a page copy when there is space in the page buffer. Our data is moved directly from DRAM to flash via the shared data bus on the DIMM. Their proactive garbage collection design, however, can improve our design as well.

5.6.2 Hybrid Memory

NVDIMMs based on emerging non-volatile memories (PCRAM, ReRAM, STT-MRAM) do not yet exist in products. Research literature has, however, explored these hybrid memory designs.

Ren et al. [86] proposed a DRAM+NVM hybrid memory design—Transparent Hybrid NVM (ThyNVM)—that periodically checkpoints to recover after a system failure. ThyNVM leveraged spatial locality of updates and determined that the working copy of sparse updates should be kept in NVM and checkpointed there via block remapping. While block remapping is possible for byte-addressable memories, it does not work for flash as we explained in Section 5.3.2.1. ThyNVM further employed block remapping to DRAM while writing back densely updated pages to NVM. This practice assumes that pages can be quickly written back to NVM because if not, this would use a lot of extra memory space; especially because densely updated pages are likely to receive the most updates.

Gao et al. [42] proposed *Mona* for hybrid DRAM+PCM systems. *Mona* writes partial checkpointing during application execution utilizing idle time periods. It divides each checkpoint interval into *dynamic partial checkpointing* (during application execution) and *final checkpointing* segments. In partial checkpointing, they estimate the *coldness* of dirty pages and write them to PCM. Our *early* checkpointing is similar to their use of idle periods. They find a sufficiently long idle period and lockdown the entire rank to perform a bulk copy of dirty pages to PCM. We do not do this because long flash program latencies are prohibitive. During final checkpointing, they finish writing the pages that they could not write back early. In our design, we employ *late*

checkpointing rather than stopping.

5.6.3 Compression

Consolidating is similar to the concept of checkpointing compression, which has been studied [50]. Moshovos et al [75] placed a compressor on-chip that compresses cache lines belonging to checkpoint records before they are sent to memory; their goal was to reduce memory bandwidth occupied by checkpoints. MCREngine [52] compresses several HDF5 format checkpoint files made by application-level checkpoints. MCREngine's compression is applied to global checkpoints written to the parallel file system (PFS). Different from existing work, we proposed a hardware-based consolidate mechanism from DRAM to flash that maintains physical page addresses.

5.6.4 Overlapping with Application Execution

Related work for hiding checkpoint overhead by overlapping with application execution falls into two distinct camps: aggressive and early or lazy and late. Early checkpointing usually involves coldness prediction or utilizing idle periods. Lazy checkpointing usually involves employing background threads, copy-on-write, and history checkpointing. Moshovos et al [75] opted for the lazy style by checkpointing the old value when it is overwritten by the new value: this is the history file method.

Yamagata et al [121] suggested a temporal reduction in checkpointing by spreading out I/O by checkpointing data as soon as their values become fixed. Between the intervals of two coordinated checkpoints, they speculatively predict whether each memory write will be the last write prior to the next checkpoint, and thus can be checkpointed early. After being checkpointed early, the page is again write protected. Any further write attempts will detect a false positive prediction. Those pages are written again at the coordinated checkpoint. Our proposed *early* checkpointing follows a concept similar to theirs.

Our idea for *late* checkpointing was inspired by [30]. They proposed BPFS—a file system for byte-addressable persistent memory. In enforcing ordering from LLC to persistent memory, rather than flushing the entire cache at each epoch boundary, BPFS required each line in the cache hierarchy to be extended by an *epoch ID counter* that tracked which epoch the cache line was modified in. The cache replacement policy would not evict a cache line from a newer epoch until all the cache lines from older epochs were evicted. Similarly, rather than stopping and checkpointing, we track the modified phase # of each page so that they could be checkpointed later on.

5.7 Summary

DIMM-based flash storage platforms are a fairly recent invention that fits an entire SSD on an interface traditionally designed for DRAM main memory. Given their functional similarity to SSDs, they are easily adopted into existing systems with minimal BIOS updates. The conventional filesystem and block-oriented access protocols that are imposed on DIMM-based SSDs, however, do not fully unlock the potential of situating the SSD on the memory bus. We designed a new communication protocol that directly moves data between main memory and flash storage without kernel intervention. Our work uses this new style of flash storage for checkpointing and proposes two new techniques—*consolidate* and *late* checkpointing—to hide flash program latency and reduce checkpointing overhead.

The proposed *consolidate* method leverages the granularity of updates to amortize the flash program latency over many page copies. It condenses sparsely updated pages and consolidates them into a single flash page. The proposed *early-late* method leverages cold periods to overlap checkpointing with application execution. The *late* method realizes that cold periods can span multiple checkpoint intervals and uses them to lazily writeback pages rather than stopping the application at the end of each interval.

Individually, our *consolidate* method and *early-late* method improved performance by 36% and 33% over stop-and-copy, respectively. Combined, they improved performance by 73% over stop-and-copy. **All in all, we reduced the average performance slowdown due to checkpointing from 2.1× to 22%**.

With regards to memory bandwidth, the consolidate method reduced the average additional memory bandwidth used by checkpointing operations from 20% in stop-and-copy to 11% with consolidation. Mispredictions in *early* checkpointing resulted in 30% of additional memory bandwidth for checkpointing, but applying consolidation reduced it to 18%.

CHAPTER VI

Conclusion

In this concluding chapter, we summarize the contributions of this dissertation and discuss their implications for future exascale system design.

We started our work by modeling a skeleton exascale supercomputer to meet 1 exaflop of performance. It had 204,800 compute nodes with 768 cores per node. The processors chosen to attain such a large number of cores needed an efficient interconnect topology that can deliver high throughput at low latencies. For that purpose, we built *Super-Star* and *Super-StarX*, two *asymmetric*, on-chip interconnect topologies that scale in performance and power towards kilo-core processors. Our best topology improved the average network latency by 45% and reduced the power consumption by 40% over the mesh topology. Asymmetric topologies decouple local and global communication and use a mix of medium-radix and high-radix *Swizzle-Switches* with the intent of matching router speed to wire speed. Unlike meshes, they can also adjust the number of high-radix global switches to be more energy proportional to the amount of traffic.

The millions of components that constitute the exascale system also make it vulnerable to their aggregate failure rate. Checkpoint/restart being the most commonly used fault tolerance mechanism, checkpointing locally to storage within the compute node was previously proposed to improve the large time overheads of traditional global

checkpointing. In the second part of this dissertation, we weighed the pros and cons of using the different types of storage platforms available at the local compute node. We did not use emerging non-volatile memory because their immature technology adds an unknown design risk to the already experimental nature of exascale system design. Instead, we proposed a hybrid DRAM-SSD checkpointing solution to achieve speed and reliability for local checkpointing while also reducing the endurance decay of SSDs. We demonstrated that for a particular set of benchmarks, we could extend the usability of the local SSD from 3 years to 6.3 years. We also proposed a dual-ECC mode for DRAM that protected both regular data and checkpoint data.

In the final part of this dissertation, we explored ways of hiding the data movement latency of checkpoint data from the main memory to the SSD even more effectively. In the proposed solution, we involved DIMM-based SSDs and designed a data movement procedure that worked with the shared memory controller to nonintrusively copy data from main memory. Our design also provided a persistence guarantee to non-blocking I/O that could not be provided in existing kernel I/O operations. The consolidation and overlapping optimizations that could be realized as a result of non-blocking I/O collectively reduced local checkpointing time overhead from 2.1× to 22%.

At a broader level, we demonstrated that architectural changes can be made to extract more performance for exascale systems. The research presented in this dissertation is pertinent to CPU architects, memory architects, storage architects, and emerging non-volatile memory architects. In future work, it would be interesting to characterize the relationship between kilo-core processors and their demand for memory bandwidth and I/O operations. If storage were to be even more closely integrated with memory, such as NVDIMM-Ps that intend to put DRAM and flash on the same module, I/O operations could be made even faster. If having non-volatile storage inside the compute node is becoming the norm, it maybe worthwhile to more tightly integrate checkpoint/restart operations.

At the same time, it is worth investigating how kilo-core processors contribute to the failure rate of supercomputers. There is still a lot of missing information regarding how supercomputers fail. As the first generation of exascale systems are designed and built, component manufacturers and system designers should integrate tools to log and analyze failures. Supercomputing institutions should collect and disseminate those failure data for analysis by researchers and industry alike. If we can determine that failure rates of individual components are high, then we can know to build components that can self-detect and self-correct failures. However, additional circuitry for error correction will certainly increase cost. On the other hand, if failure rates are relatively low, efficient checkpoint/restart solutions for node-level checkpointing like the ones proposed in this dissertation maybe sufficient.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] (2014), Netlist introduces industry's highest performing NVDIMM, http://www.netlist.com/investors/financial-news/press-release-details/2014/Netlist-Introduces-Industrys-Highest-Performing-NVDIMM/default.aspx, [Online; accessed 28-Feb-2017].

[2] (2015), Diablo's Memory1, http://www.diablo-technologies.com/diablos-memory1/, [Online; accessed 04-Mar-2017].

[3] (2016), Netlist unveils HybriDIMM storage class memory product with key industry partners, http://www.netlist.com/investors/financial-news/press-release-details/2016/Netlist-Unveils-HybriDIMM-Storage-Class-Memory-Product-With-Key-Industry-Partners/default.aspx, [Online; accessed 28-Feb-2017].

[4] (2017), AMD Opteron, http://www.amd.com/us/products/server/processors.

[5] (2017), Azul Systems Vega 3, http://www.azulsystems.com/products/vega/processor.

[6] (2017), SMART Modular Technologies Technical Brief: NVDIMM With SMARTs SafeStor™Technology, http://www.smartm.com/salesLiterature/dram/NVDIMM_technical_brief.pdf.

[7] (2017), Tilera TILE-Gx100, http://www.tilera.com/products/TILE-Gx.php.

[8] (2017), Cavium Systems Octeon, http://www.cavium.com.

[9] (2017), Top 500 List, https://www.top500.org/, [Online; accessed 09-Mar-2017].

[10] (2017), Intel Xeon E7, http://www.intel.com/content/www/us/en/products/processors/xeon/e7-processors.html.

[11] Abeyratne, N., R. Das, Q. Li, K. Sewell, B. Giridhar, R. G. Dreslinski, D. Blaauw, and T. Mudge (2013), Scaling towards kilo-core processors with asymmetric high-radix topologies, in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 496–507, doi: 10.1109/HPCA.2013.6522344.

[12] Abeyratne, N., H.-M. Chen, B. Oh, R. Dreslinski, C. Chakrabarti, and T. Mudge (2016), Checkpointing exascale memory systems with existing memory technologies, in *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pp. 18–29.

[13] Advanced Scientific Computing Advisory Committee (2015), Meeting Minutes, Crystal City, Virginia.

[14] Ainsworth, T. W., and T. M. Pinkston (2007), Characterizing the cell eib on-chip network, *IEEE Micro*, *27*(5), 6–14, doi:10.1109/MM.2007.4378779.

[15] Alverson, B., E. Froese, L. Kaplan, and D. Roweth (2012), Cray XC Series Network.

[16] Ashraf, R. A., R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose (2015), Understanding the propagation of transient errors in hpc applications, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pp. 72:1–72:12.

[17] Balfour, J., and W. J. Dally (2006), Design tradeoffs for tiled cmp on-chip networks, in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 187–198, ACM.

[18] Bautista-Gomez, L., S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka (2011), Fti: High performance fault tolerance interface for hybrid systems, in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12.

[19] Bergman, K., et al. (2008), ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead.

[20] Binkert, N., et al. (2011), The gem5 simulator, *ACM SIGARCH Computer Architecture News*, *39*(2), 1–7.

[21] Bland, B. (2012), Titan - early experience with the titan system at oak ridge national laboratory, in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 2189–2211, doi:10.1109/SC.Companion.2012.356.

[22] Bononi, L., and N. Concer (2006), Simulation and analysis of network on chip architectures: ring, spidergon and 2d mesh, in *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*, pp. 154–159, European Design and Automation Association.

[23] Borkar, S. (2007), Networks for multi-core chips–a contrarian view, in Invited Presentation to Special Session at ISLPED 2007.

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

[25] Bronevetsky, G., and R. Rugina (n.d.), Static analysis for checkpoint size reduction in array-based programs.

[26] Bronevetsky, G., D. Marques, K. Pingali, S. McKee, and R. Rugina (2009), Compiler-enhanced incremental checkpointing for openmp applications, in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, IEEE.

[27] Cappello, F., A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir (2009), Toward exascale resilience, *International Journal of High Performance Computing Applications*.

[28] Chen, R., Z. Shao, and T. Li (2016), Bridging the i/o performance gap for big data workloads: A new nvdimm-based approach, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, doi:10.1109/MICRO.2016.7783712.

[29] Chi, P., C. Xu, T. Zhang, X. Dong, and Y. Xie (2014), Using Multi-level Cell STT-RAM for Fast and Energy-efficient Local Checkpointing, ICCAD 2014.

[30] Condit, J., E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee (2009), Better i/o through byte-addressable, persistent memory, in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 133–146, ACM.

[31] Cornwell, M. (2012), Anatomy of a solid-state drive., *Commun. ACM*, *55*(12), 59–63.

[32] Das, R., S. Eachempati, A. K. Mishra, V. Narayanan, and C. R. Das (2009), Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps, in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 175–186, IEEE.

[33] Di, S., M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello (2014), Optimization of multi-level checkpoint model for large scale hpc applications, in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1181–1190.

[34] Dong, X., N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie (2009), Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems, SC 2009, doi:10.1145/1654059.1654117.

[35] Dougherty, E. (1999), *Electronic Imaging Technology*, Press Monograph Series, SPIE Optical Engineering Press.

[36] Dreslinski, R., et al. (2012), Swizzle switch: A self-arbitrating high-radix crossbar for noc systems, in *Hot Chips 24 Symposium (HCS), 2012 IEEE*, pp. 1–44, IEEE.

[37] Elnozahy, E. N., L. Alvisi, Y.-M. Wang, and D. B. Johnson (2002), A survey of rollback-recovery protocols in message-passing systems, *ACM Computing Surveys (CSUR)*, *34*(3), 375–408.

[38] Engelmann, C. (2010), Resilience Challenges at the Exascale.

[39] Finke, D. (2016), http://xitore.com/wp-content/uploads/2016/09/Introduction-to-the-NVDIMM-X-White-Paper-September-23-2016-PUBLIC-FINAL4.pdf.

[40] Gao, Q., W. Yu, W. Huang, and D. K. Panda (2006), Application-transparent checkpoint/restart for mpi programs over infiniband, in *Parallel Processing, 2006. ICPP 2006. International Conference on*, pp. 471–478, IEEE.

[41] Gao, Q., W. Huang, M. J. Koop, and D. K. Panda (2007), Group-based coordinated checkpointing for mpi: A case study on infiniband, in *Parallel Processing, 2007. ICPP 2007. International Conference on*, pp. 47–47, IEEE.

[42] Gao, S., B. He, and J. Xu (2015), Real-time in-memory checkpointing for future hybrid memory systems, in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 263–272.

[43] Gomez, L. A. B., N. Maruyama, F. Cappello, and S. Matsuoka (2010), Distributed Diskless Checkpoint for large scale systems, in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 63–72, IEEE Computer Society.

[44] Grot, B., J. Hestness, S. W. Keckler, and O. Mutlu (2009), Express cube topologies for on-chip interconnects, in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 163–174, IEEE.

[45] Grot, B., J. Hestness, S. W. Keckler, and O. Mutlu (2011), Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees, in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 401–412, ACM.

[46] Gschwind, M., H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki (2006), Synergistic processing in cell's multicore architecture, *IEEE Micro*, *26*(2), 10–24.

[47] Ho, J. C. Y., C.-L. Wang, and F. C. M. Lau (2008), Scalable group-based checkpoint/restart for large-scale message-passing systems, in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12.

[48] Howard, J., et al. (2010), A 48-core ia-32 message-passing processor with dvfs in 45nm cmos, in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108–109, IEEE.

[49] Hursey, J. (2010), Coordinated Checkpoint/Restart Process Fault Tolerance for Mpi Applications on Hpc Systems, Ph.D. thesis, Indianapolis, IN, USA.

[50] Ibtesham, D., D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell (2012), On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance, in *2012 41st International Conference on Parallel Processing*, pp. 148–157.

[51] Intel Cooperation (2012), Intel Solid-State Drive DC S3700 specification.

[52] Islam, T. Z., K. Mohror, S. Bagchi, A. Moody, B. R. De Supinski, and R. Eigenmann (2012), MCRENGINE: a scalable checkpointing system using data-aware aggregation and compression, SC 2012.

[53] Jian, X., H. Duwe, J. Sartori, V. Sridharan, and R. Kumar (2013), Low-power, Low-storage-overhead Chipkill Correct via Multi-line Error Correction, SC 2013.

[54] Johnson, D., M. Johnson, J. Kelm, W. Tuohy, S. Lumetta, and S. Patel (2011), Rigel: A 1,024-core single-chip accelerator architecture, *IEEE Micro*, *31*(4), 30–41.

[55] Kang, D., et al. (2016), 7.1 256Gb 3b/cell V-NAND flash memory with 48 stacked WL layers, ISSCC 2016.

[56] Kannan, S., A. Gavrilovska, K. Schwan, and D. Milojicic (2013), Optimizing Checkpoints Using NVM as Virtual Memory, IPDPS 2013, doi:10.1109/IPDPS.2013.69.

[57] Kao, Y.-H., N. Alfaraj, M. Yang, and H. J. Chao (2010), Design of high-radix clos network-on-chip, in *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, pp. 181–188, IEEE.

[58] Kim, J., W. J. Dally, B. Towles, and A. K. Gupta (2005), Microarchitecture of a high radix router, in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pp. 420–431, IEEE.

[59] Kim, J., J. Balfour, and W. Dally (2007), Flattened butterfly topology for on-chip networks, in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 172–182, IEEE.

[60] Kim, J., W. J. Dally, and D. Abts (2007), Flattened butterfly: a cost-efficient topology for high-radix networks, in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 126–137, ACM.

[61] Kim, J., W. J. Dally, S. Scott, and D. Abts (2008), Technology-driven, highly-scalable dragonfly topology, in *2008 International Symposium on Computer Architecture*, pp. 77–88.

[62] Kim, J., M. Sullivan, and M. Erez (2015), Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory, HPCA 2015, doi:10.1109/HPCA.2015.7056025.

[63] Kongetira, P., K. Aingaran, and K. Olukotun (2005), Niagara: A 32-way multithreaded sparc processor, *IEEE micro*, *25*(2), 21–29.

[64] Kumar, R., V. Zyuban, and D. M. Tullsen (2005), Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling, in *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 408–419, IEEE Computer Society.

[65] Lee, J. W., M. C. Ng, and K. Asanovic (2008), Globally-synchronized frames for guaranteed quality-of-service in on-chip networks, in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 89–100, IEEE Computer Society.

[66] Lin, S., and D. J. Costello (2004), *Error Control Coding*, 2nd ed., Pearson.

[67] Liu, N., J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn (2012), On the role of burst buffers in leadership-class storage systems, MSST 2012.

[68] Ludovici, D., F. Gilabert, S. Medardoni, C. Gomez, M. E. Gómez, P. Lopez, G. N. Gaydadjiev, and D. Bertozzi (2009), Assessing fat-tree topologies for regular network-on-chip design under nanoscale technology constraints, in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 562–565, European Design and Automation Association.

[69] McKeown, N. (1999), The islip scheduling algorithm for input-queued switches, *IEEE/ACM transactions on networking*, *7*(2), 188–201.

[70] Micron Technology, Inc. (2011), P320h Half-Height and Half-Length PCIe NAND Flash SSD.

[71] Micron Technology, Inc. (2013), NAND Flash Memory - MLC+ (L85C+).

[72] Micron Technology Inc. (2017), Micron NAND Flash, https://www.micron.com/products/nand-flash.

[73] Mishra, A. K., R. Das, S. Eachempati, R. Iyer, N. Vijaykrishnan, and C. R. Das (2009), A case for dynamic frequency tuning in on-chip networks, in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 292–303, IEEE.

[74] Moody, A., G. Bronevetsky, K. Mohror, and B. R. d. Supinski (2010), Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, SC 2010.

[75] Moshovos, A., and A. Kostopoulos (2004), Cost-effective, highperformance giga-scale checkpoint/restore.

[76] Ni, X., E. Meneses, and L. V. Kalé (2012), Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm, CLUSTER 2012.

[77] Objective Analysis (2015), A Close Look At The Micron/Intel 3D XPoint Memory.

[78] Ouyang, X., S. Marcarelli, and D. Panda (2010), Enhancing checkpoint performance with staging io and ssd, SNAPI 2010, doi:10.1109/SNAPI.2010.10.

[79] Passas, G., M. Katevenis, and D. Pnevmatikatos (2010), A 128 x 128 x 24gb/s crossbar interconnecting 128 tiles in a single hop and occupying 6% of their area, in *Proceedings of the 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*, pp. 87–95, IEEE Computer Society.

[80] Passas, G., M. Katevenis, and D. Pnevmatikatos (2011), Vlsi microarchitectures for high-radix crossbar schedulers, in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, pp. 217–224, ACM.

[81] Patil, H., R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi (2004), Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation, in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pp. 81–92, IEEE.

[82] Peh, L.-S., and W. J. Dally (2001), A delay model and speculative architecture for pipelined routers, in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pp. 255–266, IEEE.

[83] Plank, J. S., M. Beck, and G. Kingsley (1995), Compiler-assisted memory exclusion for fast checkpointing.

[84] Rajachandrasekar, R., A. Moody, K. Mohror, and D. K. Panda (2013), Thinking Beyond the RAM Disk for In-Memory Checkpointing of HPC Applications, *OSU Tech. Report (OSU-CISRC-1/13-TR02)*.

[85] Rajachandrasekar, R., A. Moody, K. Mohror, and D. K. Panda (2013), A 1 PB/s file system to checkpoint three million MPI tasks, HPDC 2013.

[86] Ren, J., J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu (2015), ThyNVM: Enabling software-transparent crash consistency in persistent memory systems, in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 672–685, doi:10.1145/2830772.2830802.

[87] Requena, C. G., F. G. Villamón, M. E. G. Requena, P. J. L. Rodríguez, and J. D. Marín (2008), Ruft: Simplifying the fat-tree topology, in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*, pp. 153–160, IEEE.

[88] Rubini, A., and J. Corbet (2001), *Linux device drivers*, " O'Reilly Media, Inc.".

[89] Saito, T., K. Sato, H. Sato, and S. Matsuoka (2013), Energy-aware i/o optimization for checkpoint and restart on a nand flash memory system, FTXS 2013.

[90] Samsung (2014), White Paper: Samsung V-NAND technology, http://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND_technology_WP.pdf.

[91] Sankaralingam, K., R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore (2003), Exploiting ilp, tlp, and dlp with the polymorphous trips architecture, in *ACM SIGARCH Computer Architecture News*, vol. 31, pp. 422–433, ACM.

[92] Sato, K., N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka (2012), Design and modeling of a non-blocking checkpointing system, in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pp. 1–10.

[93] Sato, K., S. Matsuoka, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinski, and N. Maruyama (2013), Burst SSD Buffer: Checkpoint Strategy at Extreme Scale, *Burst SSD Buffer: Checkpoint Strategy at Extreme Scale*.

[94] Satpathy, S., R. Dreslinski, T.-C. Ou, D. Sylvester, T. Mudge, and D. Blaauw (2011), Swift: A 2.1 tb/s 32× 32 self-arbitrating manycore interconnect fabric, in *VLSI Circuits (VLSIC), 2011 Symposium on*, pp. 138–139, IEEE.

[95] Satpathy, S., R. Das, R. Dreslinski, T. Mudge, D. Sylvester, and D. Blaauw (2012), High radix self-arbitrating switch fabric with multiple arbitration schemes and quality of service, in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 406–411, IEEE.

[96] Satpathy, S., K. Sewell, T. Manville, Y.-P. Chen, R. Dreslinski, D. Sylvester, T. Mudge, and D. Blaauw (2012), A 4.5 tb/s 3.4 tb/s/w 64× 64 switch fabric with self-updating least-recently-granted priority and quality-of-service arbitration in 45nm cmos, in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pp. 478–480, IEEE.

[97] Schroeder, B., and G. Gibson (2010), A large-scale study of failures in high-performance computing systems, *IEEE Transactions on Dependable and Secure Computing*, *7*(4), 337–350.

[98] Schroeder, B., and G. A. Gibson (2007), Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?, FAST 2007.

[99] Schroeder, B., and G. A. Gibson (2007), Understanding failures in petascale computers, in *Journal of Physics: Conference Series*, vol. 78.

[100] Schroeder, B., E. Pinheiro, and W.-D. Weber (2009), DRAM errors in the wild: a large-scale field study, in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37.

[101] Scott, S., D. Abts, J. Kim, and W. J. Dally (2006), The blackwidow high-radix clos network, in *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 16–28, IEEE Computer Society.

[102] Seiler, L., et al. (2008), Larrabee: a many-core x86 architecture for visual computing, in *ACM Transactions on Graphics (TOG)*, vol. 27, p. 18, ACM.

[103] Sewell, K., et al. (2012), Swizzle-switch networks for many-core systems, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, *2*(2), 278–294.

[104] Shahzad, F., M. Wittmann, T. Zeiser, G. Hager, and G. Wellein (2013), An evaluation of different I/O techniques for checkpoint/restart, IPDPSW 2013.

[105] Shimpi, A. L. (2010), Intel's Sandy Bridge Architecture Exposed, http://www.anandtech.com/show/3922/intels-sandy-bridgearchitecture-exposed/4.

[106] Shin, W., J. Park, and H. Y. Yeom (2016), Unblinding the os to optimize user-perceived flash ssd latency, in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, USENIX Association.

[107] Sridharan, V., and D. Liberty (2012), A Study of DRAM Failures in the Field, SC 2012.

[108] Sridharan, V., J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi (2013), Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults, SC 2013.

[109] Sridharan, V., N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi (2015), Memory Errors in Modern Systems: The Good, The Bad, and The Ugly, ASPLOS 2015.

[110] Sun, F., K. Rose, and T. Zhang (2006), On the use of strong bch codes for improving multilevel nand flash memory storage capacity, in *IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation.*

[111] Taylor, M. B., et al. (2002), The raw microprocessor: A computational fabric for software circuits and general-purpose programs, *IEEE micro*, *22*(2), 25–35.

[112] Toshiba Corporation (2017), Toshiba NAND Flash Memory, https://toshiba.semicon-storage.com/ap-en/product/memory/nand-flash.html.

[113] Udipi, A. N., N. Muralimanohar, and R. Balasubramonian (2010), Towards scalable, energy-efficient, bus-based on-chip networks, in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, IEEE.

[114] Udipi, A. N., N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi (2010), Rethinking DRAM design and organization for energy-constrained multi-cores, in *ACM SIGARCH Computer Architecture News*, vol. 38.

[115] Udipi, A. N., N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi (2012), LOT-ECC: Localized and Tiered Reliability Mechanisms for Commodity Memory Systems, ISCA 2012.

[116] U.S Department of Energy Office of Science and National Nuclear Security Administration (2014), Preliminary Conceptual Design for an Exascale Computing Initiative.

[117] Vetter, J., R. Schreiber, T. Mudge, and Y. Xie (2015), Blackcomb: Hardware-Software Co-design for Non-Volatile Memory in Exascale Systems.

[118] Wang, H., L.-S. Peh, and S. Malik (2005), A technology-aware and energy-oriented topology exploration for on-chip networks, in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 1238–1243, IEEE.

[119] Wentzlaff, D., et al. (2007), On-chip interconnection architecture of the tile processor, *IEEE micro*, *27*(5), 15–31.

[120] Xu, Q., H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan (2015), Performance analysis of nvme ssds and their implication on real world databases, in *Proceedings of the 8th ACM International Systems and Storage Conference*, p. 6, ACM.

[121] Yamagata, I., S. Matsuoka, H. Jitsumoto, and H. Nakada (2006), Speculative checkpointing.

[122] Yoon, D. H., and M. Erez (2010), Virtualized and Flexible ECC for Main Memory, ASPLOS 2010.

[123] Zhang, Y. P., T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao (2006), A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture, in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10–pp, IEEE.

[124] Zheng, G., L. Shi, and L. V. Kalé (2004), FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI, CLUSTER 2004.