
In recent years, there has been a renewed interest in near-memory processing (NMP) architectures as a workaround for the performance and energy issues of frequent and irregular memory access, which are prevalent in today’s data-intensive applications. With NMP architectures, simple yet data-intensive computations can be offloaded to near-memory processing units in order to reduce data movement across the memory hierarchy. While the idea is intuitive, programmability remains a major concern for the widespread dispersion of NMP. The lack of consensus among computer architects on the design details of NMP hardware have impeded the development of a common programming interface; moreover, even with a well-defined programming interface, a thorough understanding of the architecture’s features and limitations will be crucial in order to adapt application software to fully exploit NMP.

As this is the case, we determined that it is most appropriate to begin with redesigning fundamental, general-purpose data structures for new NMP architectures. Data structures are basic building blocks of software, yet because their implementations are encapsulated as packaged software libraries, developers can easily take advantage of highly optimized data structures without having to understand their complex implementation details. Therefore, if commonly used data structures are redesigned to optimally exploit NMP architecture, the benefits can easily extend to many applications. This dissertation focuses particularly on the NMP-aware redesign of concurrent data structures. Much of the difficulties in translating applications and algorithms to execute on NMP architecture arise from reinforcing parallelism and ensuring correct, concurrent access to data. Nonetheless, existing high-performance data structures have been optimized for high multithreaded concurrency, and NMP-based data structures must retain the concurrency and correctness guarantees in order to be useful in large-scale applications.

To this end, this dissertation conducts a holistic software-hardware co-design with conservative assumptions on the baseline NMP hardware. The first part of the dissertation work provides a thorough empirical evaluation of NMP-based data structure designs that had been verified by theoretical assumptions only. The empirical evaluation using full-system architecture simulations uncovers bottlenecks that had been overlooked by prior theoretical analysis and proposes simple hardware adjustments that reduces their impact. The second part of the dissertation work introduces and evaluates new NMP-aware algorithms for data structures that have been optimized for on-chip cache locality in conventional non-NMP systems.
Concurrent Data Structures with Near-Memory Processing: Software-Hardware Co-Design

by

Jiwon Choe

B. S. E. E., Rice University, 2013
Sc. M., Brown University, 2018

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2022
© Copyright 2022 by Jiwon Choe
This dissertation by Jiwon Choe is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________

R. Iris Bahar, Director

Recommended to the Graduate Council

Date ______________

Maurice Herlihy, Reader

Date ______________

Malte Schwarzkopf, Reader

Approved by the Graduate Council

Date ______________

Andrew G. Campbell
Dean of the Graduate School
Dedicated to my grandparents –
Jong-Byuk Choe, Seong-Sook Kang,
Ang Huh, and the late Hyuk-In Lew.
I am so blessed to have grown up with your presence in my life.
Jiwon Choe was born and raised mostly in Seoul, South Korea. She spent a few years of her childhood in Fairfax, Virginia (USA) and her high school years in Hoengseong, Gangwon (South Korea).

Jiwon received her B. S. E. E. degree from Rice University in May 2013, along with a dual B. A. degree in Computer Science. She worked in industry as a software engineer prior to joining Brown University for her graduate studies. She received her Sc. M. degree in Computer Science in May 2018, en route to her Doctoral degree.

Jiwon's research was published and presented at ACM conferences (Choe et al., SPAA 2019 and Choe et al., MEMSYS 2019), and she received the Best Student Presentation Award at MEMSYS 2019. The unpublished portion of this dissertation’s work is currently under review. Outside of research, she completed Brown University Sheridan Center’s Teaching Certificate Program on Reflective Teaching, and she served as a graduate teaching assistant for ENGN 1630: Digital Electronics Systems Design (Fall 2019).

In 2018, she was a recipient of the Cadence Women in Technology Scholarship.
Acknowledgements

First and foremost, I would like to thank my amazing advisors, Professors Iris Bahar and Maurice Herlihy. They guided me in their own ways so that I can grow as an independent researcher. They were available when I needed help, asked the right questions about my research, waited patiently so that I could develop my own ideas and thoughts, trusted the integrity and quality of my work, and had complete faith that I could make it to the end, even during periods when I myself thought otherwise. I am also thankful for our collaborator, Professor Tali Moreshet (Boston University), who was like a third advisor – being present at every weekly meeting, reading over all of my paper drafts, and providing consistent support throughout the past six years.

I would like to thank Professors Malte Schwarzkopf (Brown), Erez Petrank (Technion), Tamara Lehman (CU Boulder), and Dr. Andrew Crotty (Brown/CMU) for their substantial feedback and discussions on my work. They helped me view my work in different perspectives to find new directions to take when I felt stuck.

I am grateful for having worked with the following undergraduate and graduate students at various points throughout my Ph. D. journey: Jonathan Lister, Amy Huang, Jarod Boone, Casey Nelson, and Samuel Thomas from Brown University; Zhuohao (Jack) Yang and Ayako Shimizu from Boston University; and Ofir Gordon from the Technion. Their contributions were extremely valuable in shaping my research, whether the work made it into publications or not. I sincerely hope that they also found the experience meaningful.

I would also like to thank my family and friends. Minsik, my husband and best friend for life for everything. My parents, for their unsparing and endless support. My friends and Pastor Chu at Rhode Island Central Korean Church for their care and prayers. My small group of high school friends and college friends, for understanding all my ups and downs.

I am particularly thankful for how much my parents – especially my mom – cared that I know how to write well. It payed off so much! It feels ridiculous mentioning this in my dissertation acknowledgements, but throughout this journey I benefited so much from being able to articulate my thoughts clearly that this totally deserves a separate shout-out.

Lastly, thank you, my God, for faithfully leading me in every step and every way.
## Contents

**List of Tables**

ix  

**List of Figures**

x  

1 **Introduction**

1  

2 **Background**

2.1 Near-Memory Processing Architecture

2.1.1 Nomenclature 4

2.1.2 Advent of NMP-Enabling Hardware 5

2.1.3 Modern NMP Architecture Configurations 6

2.2 Data Structures and Architecture-Aware Optimizations

2.2.1 Data Structures Adapted for Multicore Architecture 9

2.2.2 Data Structures Adapted for Memory Access Latencies 12

3 **Related Work**

3.1 NMP-based Pointer-Chasing Acceleration 13

3.2 Locality-Aware Work-Offloading with NMP 14

4 **Baseline NMP Architecture for Dissertation Work**

4.1 High-Level Configuration 16

4.2 Simulation Framework

4.2.1 Brown-SMCSim 17

4.2.2 Background on Architecture Simulations 18

5 **Evaluation of NMP-Based Concurrent Data Structures**

5.1 NMP-Based Data Structure Design 19

5.1.1 Linked List 20

5.1.2 Skiplist 20

5.1.3 FIFO Queue 20

5.2 Hardware Modification 21
# List of Tables

4.1 Brown-SMCSim memory system configuration. .................................................. 18

5.1 Evaluation framework configuration for the empirical evaluation of NMP-based con-
current data structures. ......................................................................................... 23

6.1 Evaluation framework configuration for NMP-hybrid data structures. ............ 43
6.2 NMP operation offload costs in Brown-SMCSim. ............................................. 47
6.3 Desktop machine configuration for NMP-hybrid data structure scalability analysis. 51

7.1 Evaluation framework configuration for NMP-aware scrypt. ............................ 57
List of Figures

2.1 3D die-stacked memory layouts. .................................................. 6
2.2 Load reduced DIMM layout (Figure from [99]). .......................... 7

4.1 Baseline NMP architecture. ...................................................... 16
4.2 Brown-SMCSim NMP architecture layout. .................................. 17

5.1 NMP-based concurrent data structure design. ............................. 20
5.2 Proposed NMP vault memory controller design. ........................... 22
5.3 NMP-based linked list evaluations. ............................................ 24
5.4 NMP-based skiplist baseline evaluations. .................................... 26
5.5 Skiplist analysis with varying cache sizes. .................................. 27
5.6 NMP-based FIFO queue evaluations. ......................................... 29

6.1 Hybrid skiplist design. ............................................................. 34
6.2 Hybrid B+ tree design. ............................................................ 37
6.3 Hybrid data structure throughput with blocking and nonblocking NMP calls. ......................................................... 42
6.4 Baseline skiplist performance evaluation. .................................... 45
6.5 Hybrid skiplist energy consumption breakdown for baseline evaluations, .............................................................. 45
6.6 Baseline B+ tree performance evaluation. .................................... 47
6.7 Hybrid B+ tree energy consumption breakdown for baseline evaluations. .............................................................. 48
6.8 Hybrid skiplist sensitivity evaluation: normalized operation throughput. .............................................................. 49
6.9 Hybrid B+ tree sensitivity evaluation: normalized operation throughput. .............................................................. 50
6.10 Hybrid B+ tree sensitivity evaluation: number of memory reads. .............................................................. 50
6.11 CPU pipeline slot utilization with conventional data structures. .............................................................. 51
6.12 Breakdown of memory bound clockticks with conventional data structures. ......................................................... 51
6.13 Microarchitectural analysis of OLTP: breakdowns of CPU cycles [105]. .............................................................. 52

7.1 NMP-aware scrypt implementation. .......................................... 57
7.2 Host-based and NMP-based scrypt execution times. .................... 58
Chapter 1

Introduction

The memory wall [110], that refers to the exponentially growing gap between processor speeds and memory access speeds, has been recognized as a problem in computing since nearly three decades ago. Although single processor performance has stagnated in the past decade with the end of Moore’s law, technology has already reached a point where single processor speeds are orders of magnitudes faster than memory access speeds. Furthermore, multicore and heterogeneous architectures are continuing to increase the rate at which computations can be carried out. All in all, the memory wall seems indestructible.

In order to mitigate impacts of the memory wall, traditional computer architectures had incorporated multiple levels of cache and even data prefetchers. However, these workarounds have become less and less effective with recent advances in data-intensive applications, which include, but are not limited to, data queries and analytics over large databases, graph processing, and training and inference on various machine learning models. These data-intensive applications inherently generate frequent data access over large datasets, only to carry out few simple computations on the retrieved data. The limited temporal locality of data in these applications renders caches ineffective; furthermore, data prefetchers are often incapable of hiding memory access latencies, for some applications exhibit unpredictable memory access patterns that make prefetching impossible. Even when memory access is predictable, the sheer amount of data needed for each computation causes the low-bandwidth off-chip interconnect to become the primary performance bottleneck. Processing units end up stalling despite sophisticated prefetching capabilities because data cannot be fed into the pipeline fast enough.

To address the data movement bottleneck of data-intensive applications, computer architects have begun to revisit the idea of near-memory processing (NMP). NMP refers to the high level idea of placing processing units physically close to the memory. The physical proximity allows processing units to have high bandwidth and relatively low latency memory access, which applications can exploit by offloading data-intensive computations to these near-memory processing units. While the idea of NMP had been popular even in the 1990s [43, 64, 68, 69, 70, 86, 89], the recent explosion of data-intensive applications and timely advances in technologies that enable the realization of
NMP hardware (detailed in Chapter 2) have renewed the research community's interest in these architectures.

In the past several years, computer architects and related research communities have explored the potential of NMP architectures for a wide variety of data-intensive applications, including but not limited to: graph processing [3, 39, 83, 125, 126], data analytics [7, 39, 96], and recommendation models [65, 72]. The general consensus of these broad investigations is that NMP can be very effective for computations that are limited by memory bandwidth. The primary advantage of NMP hardware is in the high internal bandwidth provided by the tight integration of memory and logic; furthermore, the integration of memory and logic enables memory capacity-proportional scaling of aggregate bandwidth, which in turn allows for higher parallelism and scalability.

Nonetheless, programmability remains a major concern for the widespread dispersion of NMP. This is largely due to the lack of consensus among researchers on the design details of NMP architecture. Based on the application of interest, there are different perspectives on where exactly the near-memory logic should be placed, what functionality the near-memory compute units should have, and how access boundaries should be defined for data stored in NMP-capable memory. For example, some proposals suggest tightly coupling a compute unit with a small private portion of memory; some suggest having several compute units with shared access to a relatively large portion of memory; some suggest that even on-chip host processors should share access to NMP-capable memory. Regardless of the rationale behind each proposal, without consensus on the architecture-level design details, a common programming interface cannot be developed. Moreover, even with a programming interface set in place, in order to fully exploit the new NMP architecture, application developers will need to first understand the architecture's features and limitations, then determine how the application should partition its data, decide what portions of its computation can be offloaded to near-memory compute units, and figure out how concurrent access to data should be managed. However, none these are trivial tasks.

As this is the case, we determined that it is most appropriate to begin with redesigning fundamental, general-purpose data structures for new NMP architectures. Data structures are basic building blocks of software, yet in software development, they are simply imported as packaged libraries and used with simple API function calls, such as \texttt{insert(x)}, \texttt{read(x)}, or \texttt{remove(x)}. The encapsulation allows application developers to take advantage of highly optimized data structures without having to understand their complex implementation details. Therefore, if commonly used data structures are redesigned to optimally utilize the NMP architecture, the benefits of NMP can easily extend to many applications. In fact, this had been the case with novel architectures in the past. When multicore architectures emerged, data structure libraries were updated with new algorithms that ensure concurrent yet correct data structure access and manipulation [1, 5, 26, 30, 46, 57, 74, 113]. When Non-Uniform Memory Access (NUMA) architectures emerged, new NUMA-aware data structure algorithms were developed to reduce relatively expensive remote memory accesses [19, 31].

In this work, we are particularly focused on redesigning concurrent data structures for NMP architectures. Much of the difficulties in translating applications and algorithms to execute on NMP
architecture arise from reinforcing parallelism and ensuring correct, concurrent access to data. Despite the challenge, this is an important area of work, for conventional high performance systems – where NMP-capable memory devices are likely to be incorporated – are fundamentally multicore systems. As described earlier, data structures for multicore architectures have been heavily optimized to achieve maximal concurrency, which is central to high performance. For NMP-based data structures to be useful and widely adopted, they must provide even higher performance than these existing concurrent data structures; to this end, they must retain high concurrency, while also exploiting NMP’s features and working around its limitations.

A holistic software-hardware co-design is needed in order to provide an optimal solution to the key research question of this dissertation: how should NMP-based concurrent data structures be designed? To this end, this dissertation brings in expertise from both computer architecture and data structure algorithm design. More specifically, the first part of this dissertation work (Chapter 5) provides a thorough empirical evaluation of simple NMP-based concurrent data structure designs, using full-system architecture-level simulations. The evaluation shows that prior theoretical analysis of NMP-based data structures had been overly optimistic, for prior analysis had overlooked effects of DRAM activity and on-chip cache locality. Nonetheless, our work proposes simple hardware and software enhancements that allow NMP-based data structures to still achieve high performance and low energy consumption. The second part of this dissertation work (Chapter 6) focuses particularly on adapting hierarchical data structures, that benefit from high on-chip cache locality on conventional non-NMP systems, to NMP-aware designs.

On another note, a tangential research problem was also briefly explored during the course of work for this dissertation. While the memory wall is generally considered a phenomenon that negatively impacts application performance and energy consumption, the non-negligible time and energy costs of conventional DRAM-based memory access are sometimes used to its advantage for security purposes. In fact, memory-hard cryptographic functions \footnote{memory-hard cryptographic functions} compute password hashes via a long chain of pseudo-random memory accesses in order to provide protection against brute-force security attacks. The last part of this dissertation (Chapter 7) presents the background and results from preliminary investigations on how NMP may impact the security of such memory-hard functions.

This dissertation is organized as follows. Chapter 2 covers the background behind (1) NMP-enabling hardware and (2) generic data structure optimizations. Chapter 3 reviews prior work that are particularly closely related to research done for this dissertation. Chapter 4 gives an overview of the baseline NMP architecture used in our work. Chapters 5 and 6 elaborate on the dissertation work described above. Chapter 7 covers the preliminary investigation into how NMP may impact the security of memory-hard cryptographic functions. Chapter 8 discusses future research directions and concludes the dissertation.
Chapter 2

Background

2.1 Near-Memory Processing Architecture

2.1.1 Nomenclature

At a high level, near-memory processing (NMP) architectures refer to architectures in which processing units are placed physically close to the memory. The primary goal of NMP architectures is to exploit high internal memory bandwidth and relatively low memory access latency provided by the physical proximity of compute to memory. In this dissertation, I will generically use the terms NMP cores to refer to near-memory processing units and NMP-capable memory to refer to memory that is equipped with NMP cores. Host or host-side is used to refer to the system-on-chip.

NMP has appeared in prior work under many different names: near-memory computing, near-data-processing, intelligent RAM, processing-near-memory, or processing-in-memory, just to name a few. While near-memory computing or processing-near-memory unmistakably have the same meaning as NMP, other names are ambiguous. Intelligent RAM (IRAM) was specifically a term used in the late 1990s to refer to NMP hardware developed at the time. Near-data-processing (NDP) is often used as a broader term that encompasses all architectures in which processing units are placed near where data resides. This includes not only near-memory processing but also near-cache or near-storage processing architectures. Processing-in-memory (PIM) has various meanings as well. Although a significant number of works use PIM to refer to NMP, PIM is also used frequently to mean processing-using-memory, which is a different type of architecture that leverages memory circuitry itself to execute simple bitwise or arithmetic operations directly in the memory array. Furthermore, a similar-sounding term in-memory processing is commonly used in the database community to refer to storing and processing databases entirely in main memory, as opposed to reading from and writing to disk to store and process databases.
2.1.2 Advent of NMP-Enabling Hardware

Because the memory wall had been a known issue since the 1990s, many computer architects at the time had already suggested integrating memory and logic on the same chip (e.g., the EXECUBE chip [68, 70], the DIVA system incorporating EXECUBE chips [43], Active Pages [86], the Berkeley IRAM [90], and the FlexRAM project [64]) as a means to bridge the memory-processor performance gap. In particular, many of these proposals pointed out that significant amounts of available memory bandwidth were wasted because of separate processor and memory chips in conventional computer organizations. The NMP solutions devised at the time were therefore focused on extracting high DRAM bandwidth, which is not unlike the purpose of more recent NMP architecture proposals.

Unfortunately, the lack of commercial interest in NMP back then had lead the academic research to also die out. As mentioned in the Berkeley IRAM paper [90], the semiconductor industries for microprocessors and memory had traditionally been divided because the two types of devices had different requirements and optimization points. Microprocessor fabrication aimed for faster logic, while memory fabrication pursued smaller DRAM cells and lower leakage current to increase density and to reduce DRAM refresh rates. Moreover, the memory industry was focused on low-cost fabrication and standardization of the memory interface. A 2012 reflection paper [109] on the FlexRAM project [64] noted that considering such circumstances, the intrusive and radical idea of integrating computational logic onto memory chips would not have been appealing to the memory industry at the time.

However, since then, there have been advances in chip fabrication and packaging technology that may at last enable the commercial manufacturing of NMP hardware. Three-dimensional (3D) chip integration technology has developed out of the need to scale in terms of density, performance, and power in the face of the end of Moore’s law and Dennard scaling [71]: Through-Silicon Via (TSV) has developed as a technology for efficient vertical connections in the 3D chip stack [99]. More specifically, TSV interconnects are fabricated as holes etched into the device wafer and filled with metal, forming short permanent metal connections between chips stacked and packaged into a single device. Such physical properties of TSV interconnects provide low electric resistance and capacitance compared to long wire connections, allowing for dense connections and high data rates at low power and high reliability.

Although these developments were not made for the purpose of actualizing NMP, they surfaced as enablers for commercial NMP devices and a promising solution to the ever-exacerbating memory bandwidth issue. Hybrid Memory Cube (HMC) [56] and High Bandwidth Memory (HBM) have been announced as new memory devices that stack several DRAM dies on top of a logic die, to which high memory bandwidth is provided via TSV interconnects. The HMC and HBM specifications currently utilize the logic die in the memory stack for memory controller and interface logic only, but the potential to implement additional computational logic sparked the renewed interest in academic NMP research.

On another note, some recent NMP proposals [65, 72] have been built on commodity DRAM modules not based on 3D-stacked memory. Load Reduced Dual Inline Memory Modules (LR-DIMMs,
also referred to as buffered DIMMs\(^1\) are behind these new NMP proposals. LR-DIMMs are similar in form to typical memory modules, but they are augmented with buffers that latch the command/address and data lines of the device. While the primary purpose of LR-DIMMs is to reduce capacitive load on memory buses, which in turn allows for higher memory capacity, more recent NMP proposals have leveraged the buffer area of LR-DIMM to implement near-memory logic.

### 2.1.3 Modern NMP Architecture Configurations

This subsection reviews architecture specifics of the baseline NMP-enabling hardware introduced in Section 2.1.2 and discusses how they have impacted the NMP implementations.

Figure 2.1: 3D die-stacked memory layouts. Unused chip area on the bottom logic die is exploited to implement near-memory logic.

**NMP architecture based on 3D die-stacked memory.** Figure 2.1a shows a detailed layout of HBM (Figure from [61]). TSVs connect memory banks in the stacked DRAM dies to the bottom logic die, which is composed of buffers, testing logic, and interface logic for connections with the host. Figure 2.1b from HMC specifications [56] depicts the basic layout of HMC. A difference from HBM is that HMC is divided into vertical sections called *vaults*. On the logic die, in addition to the testing and interface logic, there is a *vault controller* for each vault, which is a memory controller that manages accesses to DRAM banks in the DRAM dies stacked above it. TSV interconnects provide high bandwidth memory access between the vault controller and DRAM banks within the vault. In both 3D die-stacked memories, there is unused space on the logic die that can be exploited to implement near-memory logic, which we generally refer to as NMP cores. In implementing NMP cores, sophisticated hardware typically applied in host processors for high performance – such as

\(^1\)Although precisely speaking, buffered DIMMs include registered DIMMs (R-DIMMs) that latch only command/address lines, which cannot be exploited for NMP.
large caches – are mostly avoided in order to fit the limited die area and power budget. (This later holds true for buffered DIMM-based NMP implementations as well.)

Many HMC-based NMP architectures have proposed implementing per-vault NMP cores, each accessing data only in their own respective vaults [3, 4, 52, 96, 100, 118, 119, 120, 125, 126]. This simple and straightforward assumption removes the need for complex hardware or software to manage data races or data coherence among NMP cores, while it still allows for high parallelism and aggregate memory bandwidth across the multiple vaults (according to HMC specifications [50], each cube has 32 vaults, and multiple cubes can be interconnected to provide more memory capacity). In fact, this restricted NMP architecture can be sufficient for embarrassingly parallel applications like MapReduce that provide simple data-partitioning and high memory-level parallelism with little modification in the software. However, the constraints are problematic for other applications that cannot partition data so simply. Most notable of such applications would be graph processing, where graph edges can span across graph nodes in different vaults. In order to support such applications, NMP cores need mechanisms to pass work and/or data to one another; moreover, the applications need to be redesigned so that communication overhead is minimized.

Other NMP architectures – either HMC- or HBM-based – have suggested NMP cores with shared access to all memory within the 3D-stack [54, 55, 59, 91, 111, 124] with NMP cores exploiting high aggregate memory bandwidth. Under this assumption, various near-memory logic have been proposed, ranging from specialized accelerators [54, 59] to general-purpose CPUs [111] and even simple GPUs [55, 91, 124]. While this seemingly provides more flexibility in the applications that could be accelerated with NMP, complex software-hardware mechanisms need to be adopted to manage shared data access and coherence among the NMP cores.

Figure 2.2: LR-DIMM layout (Figure from [99]). The buffer chip area on the DIMM is leveraged to implement near-memory logic.

![LR-DIMM Layout](image)

**NMP architecture based on LR-DIMM.** Figure 2.2 depicts the layout of LR-DIMM and how
LR-DIMMs are connected within a system. Memory is provided through multiple DIMMs plugged into the system, and the buffer chip for latching registers is located on each DIMM. This naturally translates to per-DIMM NMP logic, which is very similar to the per-vault NMP configurations based on HMC. NMP cores implemented on the DIMM’s buffer chip are provided with much higher memory bandwidth compared to host-side processors or accelerators, for it is often the off-chip data interconnect that significantly limits memory bandwidth. However, the downside is that passing work or data across different DIMMs is nontrivial in the per-DIMM NMP configurations, even more so than in the per-vault NMP configurations with 3D die-stacked memory. Recent buffered DIMM-based NMP proposals [65, 72] have simply worked around this limitation by focusing on a single particular application (most notably deep learning recommendation model inference [84]) that allows for complete data partitioning.

Other NMP architecture configurations. There have been NMP architectures that propose more fine-grained NMP core placement. UPMEM [32] is an NMP-capable DDR4 DIMM that integrates a data processing unit with every 64MB DRAM bank. Samsung also proposed NMP architecture based on HBM that exploits DRAM bank-internal bandwidth and bank-level parallelism [73], instead of an architecture that exploits the high bandwidth provided to the logic die via TSVs.

Nonetheless, most NMP architecture proposals in recent literature share the following assumptions: (1) NMP-capable memory is divided into partitions (whether the division is by vaults, DIMMs, or DRAM banks), and (2) each partition is tightly coupled with an NMP core, which has exclusive access to data in the coupled partition. The baseline NMP architecture adopted in this dissertation work also follows these assumptions. The details of the architecture model will be discussed further in Chapter 4.

NMP architecture configurations with respect to host processors. Another point of concern in NMP architecture is whether the NMP-capable memory should be shared with host processors, and if so, how coherence should be maintained on the shared data. Many works proposed either making the NMP-capable memory uncacheable on the host [3, 65, 72, 73] or dividing the computation so that host processor and NMP core execution never overlap [8, 59, 39, 96]. Other works that allow host processors and NMP cores to have concurrent access to shared data often specifically focus on the problem of shared data access, proposing host-NMP coherence mechanisms [17] or mechanisms to ensure that computation is carried out where the data resides [4, 63, 55, 79, 91, 111].

Lastly, virtual address translation from the NMP cores is also a design concern. Virtual address translation resources available in the host processors cannot be utilized efficiently by NMP cores, yet if NMP cores are to use the same virtual addresses, their translation tables must be coherent with the host-side ones. Proposals where NMP-capable memory is accessible only by NMP cores avoid this issue by having NMP cores operate on physical memory addresses [65, 72]. Other solutions propose using large pages [52, 39, 59] or compact region-based page tables by placing contiguous physical pages in contiguous virtual address space as much as possible [8, 54, 73, 78, 100, 117]. They also assume that page swapping rarely happens in the target applications or explicitly employ page locking.
This dissertation work is based on simple and generic NMP hardware so that we can focus specifically on NMP-enabled data structure designs and the necessary hardware support. We assume that each NMP-capable memory partition is accessed only by the coupled NMP core, thereby eliminating the challenges of data coherence and virtual address translation.

2.2 Data Structures and Architecture-Aware Optimizations

Data structures are fundamental building blocks of modern software; software developers choose to use certain data structures depending on how data needs to be represented and is used in an application. To name a few data structures, stacks or FIFO queues are used to quickly insert and remove items based on their insertion order. Hash maps are used to quickly find items based on a key. Sorted trees and optimized equivalents (e.g., B+ trees, skiplists) are also used to quickly find items based on a key. The lookup times with these data structures may be slower than hash maps, but they have an additional capability of providing data items in sorted order, according to their keys. Linked lists are used to store a sequence of data items; a linked list represents data items as nodes connected via pointers. Hence, insertions and removals of items at any location of the list can be processed easily. The linked list is also often the building block of other data structures, including the ones mentioned above.

Nonetheless, the details of each data structure’s implementation are often hidden from an application developer. Data structures are usually provided as packaged software libraries, and simple API function calls are used to access and modify the data structure. For example, insert(x) inserts a new item x into the data structure; read(x) finds and retrieves the data associated with x; and remove(x) removes the item x. Because this is the case, application developers only need to be aware of each data structure’s high-level construction and its appropriate usage.

Yet, actual implementations encapsulated in the packaged libraries are quite complex. Data structures are heavily optimized to provide high overall performance, and oftentimes this involves adapting the data structures to the underlying hardware architecture. This is particularly important when major changes are incorporated into the underlying architecture. Data structures need to be carefully redesigned to fully exploit new hardware features and to work around newly introduced limitations. In this section, we look into data structure adaptations driven by the shift to multicore architecture and the ever-increasing impact of memory access latency, in order to provide background on factors that must be considered as we redesign concurrent data structures for NMP architecture.

2.2.1 Data Structures Adapted for Multicore Architecture

Prior to the advent of multicore architectures, an application’s execution time depended heavily on its algorithmic complexity. In turn, data structures used in applications were designed with the goal of achieving lower theoretical bounds under single-threaded execution. These single-threaded

\[\text{Data structures that are constructed with nodes connected to one another via pointers are collectively called} \] pointer-chasing \text{data structures.}\]
algorithmic improvements, along with continually increasing processor speeds, allowed for faster data structure operations and contributed to shorter application runtimes.

However, application speedup in multicore architectures depends heavily on fully exploiting the parallelism provided by multiple processor cores. To attain high parallelism, applications must utilize multiple threads, and accordingly, data structures used in the application must be carefully redesigned for correctness under concurrency. Such redesigned data structures are called concurrent data structures. A concurrent data structure must guarantee that (1) concurrent operations are applied on the data structure as intended and (2) operations eventually complete.

**Coarse-grained synchronization** is the most straightforward way to implement a concurrent data structure. This simply involves adding a lock to a single-threaded data structure design. Every operation on the data structure would first acquire the lock (or wait on the lock if the acquisition fails), apply the operation according to the single-threaded algorithm, and then finally release the lock. Although coarse-grained synchronization is easy to reason about, it allows only one thread to actually access and/or modify the data structure at a time. This is highly problematic if the data structure is frequently accessed by a large number of threads, and even more so if each of the data structure operations involve long execution times, which is often the case with pointer-chasing data structures such as linked lists.

One important observation that concurrent data structures can leverage is that read-only operations can safely execute concurrently without affecting the correctness of one another; only operations that incur writes and modifications to the data structure have impact on correctness. **Reader-writer locks** can be used with coarse-grained synchronization to leverage this observation. A concurrent data structure protected by a reader-writer lock allows multiple read-only operations to run in parallel, but when a write-involved operation begins to execute, all other threads are prevented from reading or modifying the data structure. The reader-writer lock’s internal implementation ensures that a writer thread eventually acquires the exclusive lock to make progress.

While the use of a reader-writer lock can significantly improve performance, especially with read-heavy workloads, concurrent data structures can be further parallelized for even write operations. With coarse-grained reader-writer synchronization as described above, a write operation in its entirety executes with exclusive access to the data structure. However, in many cases, only a small part of a write operation actually modifies the data structure. For example, a linked list’s insert operation can be decomposed into two phases: the traversal phase and the actual write phase. The traversal phase finds the position at which the operation should take place, and this takes \( O(n) \) execution time. Yet, this phase is read-only and therefore can run concurrently with other read-only operations. The write phase is when pointers are updated to link the new node into the list. This phase needs to be carefully coordinated with other threads, but it takes only constant time. Furthermore, even write phases of concurrent operations can run simultaneously if they access and modify different parts of the data structure. In this regard, **fine-grained synchronization**, which decomposes the data structure into components that can be individually synchronized, provides further parallelization. Again, in the linked list example, locks can be placed on individual
nodes rather than on the entire list in order to coordinate concurrent access and modifications at every position of the list separately.

Yet, fine-grained synchronization still has inefficiencies. Ideally, high levels of parallelism can be achieved, but the simple act of acquiring and releasing locks (even with reader-writer locks) incurs high volumes of data bus traffic for synchronization among threads [28]. This can lead to degraded performance, especially if locks need to be continuously acquired and released for each operation. Optimistic synchronization, largely enabled by lock-free synchronization and various lazy synchronization methods, addresses this issue and allows for highly scalable concurrent data structures. Optimistic synchronization is based on the idea that “bad things rarely happen.” As for lock-free synchronization, a concurrent data structure is formally defined lock-free if some function call on it always finishes in a finite number of steps [48, 49]. In its implementation, atomic operations like compare-and-swap (CAS) are used in order to apply modifications to the data structure [34, 36, 47, 82]. A CAS failure indicates that a concurrent modification has been applied on the same part of the data structure, in which case the function call involving the modification is retried from an earlier step (or even from the beginning). In theory, this may cause an operation to never complete, but in reality, contention rarely happens, and retrying significant portions of the data structure operation in the rare cases have little impact on overall performance. Moreover, synchronization traffic is generated only when modifications are made on the data structure, making lock-free data structures more performant than lock-based ones in general.

In order to ensure correctness, lock-free data structures often incorporate logical deletion [36, 44, 45] in their algorithms. With logical deletion, nodes or data items that are to be removed from the data structure are first simply “marked” as deleted, and their physical removal are gradually applied. Logical deletion is an example of lazy synchronization, which is based on the idea that high-overhead work can be postponed and completed off the critical path as long as a data structure operation always produces a logically correct output.

There are also cases where logical deletion is applied not necessarily for lock-free correctness but simply to postpone and avoid performance overheads associated with synchronizing structural changes to the data structure – particularly in tree structures [60, 85, 102, 123]. Other lazy synchronization mechanisms are adopted for similar reasons. To name a few examples from recent literature, the no-hotspot skiplist [29] lazily adjusts the skiplist node heights in order to achieve ideal performance; a NUMA-optimized skiplist [31] that maintains separate upper levels for each NUMA node also applies updates to the upper levels in a lazy manner.

---

3As opposed to wait-free, where every function call finishes in a finite number of steps.

4Non-Uniform Memory Access architectures, where multiple CPU chips and memory chips are interconnected in a way such that a processor’s access to local memory is faster than its access to remote memory. The co-located CPU and memory chips are often referred to as NUMA nodes.
2.2.2 Data Structures Adapted for Memory Access Latencies

Data structure optimizations have also focused on avoiding performance degradation caused by costly memory accesses, even in pointer-chasing data structures. At a high level, basic data structures such as hash maps or balanced trees already provide $O(1)$ or $O(\log N)$ theoretical bounds on node traversals, but more complex optimizations aim for even shorter traversals and/or higher spatial and temporal locality to better exploit the cache. This subsection introduces some of such optimizations.

Splay trees [107] and concurrent variants like the CBTree [11] focus on enhancing temporal locality and achieving shorter traversal across long-term execution by moving frequently-accessed data to closer to the tree root. The biased skiplist [9] or the splay-list [5] are similar ideas for the skiplist – readjusting node heights based on each node’s access frequency. Ideally, the extra computation involved in dynamically restructuring the data structures is amortized by the overall reduction in memory access latencies.

Other optimizations focus on enhancing spatial locality of the data structures. Cohen et al. [26] note that higher levels of a typical tree structure modified much less frequently than its lower levels, and they exploit this fact to instead compact the upper levels together for better spatial locality. Similarly, many cache-conscious data structure designs increase the node size or compact data items together into a contiguous memory chunk in order to achieve better spatial locality (e.g., B+ tree [11], skip vector [18], locality-aware linked list [98]). There are also data structures custom-designed for high cache locality, such as the Masstree [80].

Some data structure designs adapted for memory access latencies have emerged with the advent of new memory technologies. NUMASK [31] (mentioned in Section 2.2.1) and Node Replication techniques [19] are examples of data structure designs optimized for NUMA architectures; these focus on avoiding expensive remote memory access and synchronization traffic across NUMA nodes. FPTree [87] is an example of a persistent B+ tree designed for non-volatile memory (NVM): FPTree suggests using an NVM-DRAM hybrid architecture to construct upper levels of the tree in DRAM and to persist only the bottommost level containing the actual data in NVM, in order to avoid the performance and write-wearout issues of NVM.
Chapter 3

Related Work

This dissertation’s focus – the hardware-software co-design of high-performance concurrent data structures for NMP architectures – has similarities with prior work in the following areas:

1. NMP-based acceleration of pointer-chasing data structure operations and applications, and
2. locality-aware work-offloading to near-memory processing units.

Each is closely related to the first and second parts of this dissertation work, respectively.

3.1 NMP-based Pointer-Chasing Acceleration

Hong et al. [52] proposed linked list traversal (LLT) engines as near-memory processing units. They acknowledged that naïve offloading of pointer-chasing operations may hurt performance; they thus devised partitioning and batching schemes that increase parallelism among LLT engines and reduce idle time in the host processors. However, their schemes leverage operation characteristics very particular to a small set of applications that they focus on (e.g., Memcached, Hash join) and are hard to generalize.

IMPICA [54] is a near-memory pointer-chasing accelerator that can be used for generic pointer-chasing workloads. The work is motivated by two challenges: (1) latencies in memory access are inevitable, and (2) near-memory accelerators cannot easily access memory management units of host CPUs, making virtual address translation nontrivial. IMPICA addresses these challenges by using a page table based on contiguous regions of virtual and physical memory and keeping multiple memory requests in-flight. The IMPICA engine handles the address translation of successive memory reads while waiting for memory accesses to complete.

Santos et al. [100] extended the ISA of the near-memory accelerator from their prior work [101] to include a FIND instruction, which launches the data structure traversal needed in order to find the item specified in the instruction. Instead of dividing and placing data structures so that a traversal is contained within a single partition, the per-partition pointer-chasing engines forward FIND operations to one another when a remote partition needs to be accessed.
A lot of recent work in NMP focus on graph processing applications, whose pointer-chasing aspect may seem similar to certain data structure operations. However, graph processing applications’ data traversal patterns and computations performed on each node are significantly different from data structure operations, limiting the relevance to one another. That being said, Tesseract [3] is one of the earliest works exploring graph processing with NMP enabled by 3D die-stacked memory. The work shows that memory bandwidth is the primary bottleneck of graph processing workloads, and they propose a near-memory graph accelerator design that fully utilizes the available memory bandwidth and effectively hides communication overhead among the near-memory cores. GraphPIM [83] takes advantage of near-memory atomic operations provided with the HMC specifications to offload graph processing operations at instruction granularity. GraphH [30], GraphP [125], and GraphQ [126] are all particularly focused on partitioning the graph data effectively and providing hardware and software support to minimize data movement and communication overhead among near-memory graph processing units.

3.2 Locality-Aware Work-Offloading with NMP

Ahn et al. [4] proposed locality-aware offloading of NMP operations at instruction granularity. Regions of code marked as NMP-offload candidates are compiled into PIM-Enabled Instructions (PEI), which can be executed either on the host-side PEI Computation Unit (PCU) or on the memory-side PCU. A PEI Management Unit in the last-level cache monitors the locality of data used by PEI, in order to determine at runtime where to execute the PEI.

CAIRO [42] is another work that suggests locality-aware instruction-level offloading to NMP. Whereas PEI required the programmer to first identify NMP-offload candidates (even though the actual offload decision was made at runtime), CAIRO identifies the NMP-offload candidates without user input and instead makes the offloading decision at compile time.

Tsai et al. [111] proposed a thread scheduling algorithm for NMP-incorporated systems. The scheduling algorithm allowed threads to be mapped to host processors or to near-memory cores dynamically, based on per-thread cache miss rates and on-chip cache resource contention. The algorithm design leveraged ideas from cache partitioning problems in literature. Hardware monitors were used in the work to dynamically profile the cache miss rates and resource contention.

Lockerman et al. [79] and Kandemir et al. [63] went a step further and suggested locality-aware near-data processing throughout the entire memory hierarchy. Lockerman et al.’s Livia architecture places Memory Service Elements (MSEs) at every component in the memory hierarchy, including at each cache level. Computations are offloaded to appropriate MSEs at task granularity, which are defined by a programming model. The offload decision is made at runtime, based on where the data resides. To ensure that frequently accessed data eventually moves up the hierarchy, MSEs would simply fetch remote data to their local memory with probability 1/32. (In PEI [4], moving data up the hierarchy was determined by a complex locality-monitoring mechanism.)
Kandemir et al.’s baseline Near-Data Computing (NDC) architecture similarly assumes that compute capabilities are available at cache controllers, memory controllers, within memory banks, and on network-on-chip link buffers. They observe that performance improvement with NDC depends on all operand data being available at the near-data compute unit within a small time delay; long operand wait-times would cause additional delay and wasted hardware resources. Based on this observation, they propose new compiler algorithms that make appropriate tradeoffs to optimally offload operations to near-data compute units.

On another note, Gokhale et al. [40] enhanced the host cache locality of data structure operations by using NMP to dynamically restructure scattered data into a cache-friendly layout.
Chapter 4

Baseline NMP Architecture for Dissertation Work

4.1 High-Level Configuration

In this dissertation, we assume a simple and generic NMP architecture. This is in order to find out the minimal hardware requirements and broadly applicable algorithmic details for concurrent data structures to be effectively implemented with NMP architecture.

Figure 4.1 outlines our baseline architecture. We assume that a system’s memory consists of host-accessible main memory without any NMP capabilities and a separate NMP-capable memory. As discussed in Chapter 2.1.3 restricting host processor access to NMP-capable memory eliminates many challenges that NMP architectures need to address, including data coherence and virtual address translation across host processors and NMP cores. The NMP-capable memory is further divided into physically separate partitions (referred to as NMP partitions or NMP vaults; terms used interchangeably throughout the dissertation). Each of these partitions is coupled with an NMP core that has exclusive access to data in the partition, and each NMP core is modeled as an in-order, single-cycle processor without any cache. The NMP core is instead equipped with a small scratchpad memory for storing program instructions and the program stack.

![Baseline NMP architecture for dissertation work.](image)

Figure 4.1: Baseline NMP architecture for dissertation work.
4.2 Simulation Framework

Because NMP-capable memory is not available as off-the-shelf devices yet, researchers currently rely on architecture simulations to evaluate their work\(^1\). This dissertation uses Brown-SMCSim [22], a full-system NMP architecture simulator built with gem5 [13]. Just like other cycle-accurate architecture simulations, Brown-SMCSim accurately models cycles and latencies associated with every hardware event.

4.2.1 Brown-SMCSim

Brown-SMCSim is extended from Azarkhish et al.’s SMCSim [8]. SMCSim originally modeled an NMP architecture based on one HMC device, with only a single NMP core (simple in-order processor) on the logic die accessing all HMC vaults. Moreover, all HMC vaults were used as host-accessible main memory; in other words, the host processors and the NMP core had shared access to memory. Nonetheless, we chose to extend SMCSim because it was a solid simulator based on HMC hardware configurations, and it provided a well-defined software stack for the host-NMP interface.

Significant modifications were made on SMCSim to simulate a system that conforms to the architecture depicted in Figure 4.1. Major modifications included (1) dividing the HMC vaults to host-accessible main memory and NMP-capable memory (the latter inaccessible by host processors), (2) creating an NMP core for each NMP vault and rearranging the components such that each NMP core is tightly coupled with an NMP vault with appropriate memory access procedures, and (3) extending the host-NMP interface software stack (e.g., drivers, API) to support multiple NMP cores in the system. Figure 4.2 depicts the HMC-based NMP architecture modeled in Brown-SMCSim, and Table 4.1 provides details on the memory system.

---

\(^1\)Although UPMEM [32] is marketed as the first real NMP hardware, it has not yet been commercialized as an off-the-shelf product.
Table 4.1: Brown-SMCSim memory system configuration.

<table>
<thead>
<tr>
<th>Memory configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vaults</strong></td>
</tr>
<tr>
<td>128MB/vault (total 2GB), 8 DRAM banks/vault</td>
</tr>
<tr>
<td><strong>DRAM parameters</strong></td>
</tr>
<tr>
<td>$t_{RP}$: 13.75ns, $t_{RCD}$: 13.75ns, $t_{CL}$: 13.75ns, $t_{BURST}$: 3.2ns</td>
</tr>
<tr>
<td><strong>NMP cores</strong></td>
</tr>
<tr>
<td>40kB scratchpad memory/NMP core:</td>
</tr>
<tr>
<td>- stores instructions and program stack</td>
</tr>
<tr>
<td>- 8kB reserved for host memory-map, used for communication with host</td>
</tr>
</tbody>
</table>

4.2.2 Background on Architecture Simulations

A full-system architecture simulation models all hardware components of a complete system, including host processors, caches, interconnects, memory devices, and relevant peripherals. Because gem5 is a modular and configurable architecture simulation framework, new hardware components (in our case, NMP-capable memory with coupled NMP cores) can be created and incorporated into the simulated full-system with appropriate hardware parameters. Moreover, the simulated system runs a real operating system, device drivers, and the full software stack in order to execute an application of interest, providing an evaluation setting very close to a real system.

However, the downside of simulator-based evaluations is that real execution time (as opposed to simulated execution time, which is the execution time that matters in terms of evaluation results) gets extremely slow. gem5 is an event-driven simulation framework, meaning that the system execution is simulated and driven forward by processing scheduled hardware events and scheduling new events derived from each processed event. The more hardware components a simulation consists of, the slower its real execution time will be, for more hardware events will be generated and processed in the simulation. The same applies with simulating more sophisticated hardware components (e.g., pipelined, out-of-order CPUs). Therefore, simulated systems are often smaller in scale compared to real high-performance machines in order to complete experiments in a reasonable time frame. For example, SMCSim and Brown-SMCSim modeled only 16 vaults for the HMC device (rather than 32 as defined in the HMC specifications).
Chapter 5

Evaluation of NMP-Based Concurrent Data Structures

An initial theoretical analysis described how the flat-combining (FC) synchronization scheme can be used in NMP cores to manage concurrency in NMP-based linked lists, skiplists, and FIFO queues. The first part of my dissertation work focused on the empirical evaluation and detailed analysis of these data structures on a full-system NMP architecture simulator.

5.1 NMP-Based Data Structure Design

In a non-NMP context, a flat-combining data structure consists of a publication list and a combiner lock. Each thread posts its operation request, such as contains(x), add(x), remove(x), in a dedicated slot in the data structure’s publication list. After posting the request, threads compete for the combiner lock. The thread that acquires the combiner lock becomes the combiner thread, and it is the only thread that accesses the actual data structure. The combiner thread goes through the publication list, executes the posted operations, writes return values to corresponding slots in the publication list, and releases the combiner lock. Other threads that fail to acquire the combiner lock simply spin on their own slots in the publication list until the return value is set, which indicates that the operation has been applied to the data structure.

Flat-combining is well-suited to NMP-based concurrent data structures. The data structure itself resides in one or more NMP vaults, and each NMP core acts as a dedicated combiner thread for the portion of the data structure contained in its coupled NMP vault. Host processor threads simply send data structure operations to the corresponding NMP core; each NMP core combines and reorders concurrent operations that it receives and executes the operations on behalf of the host threads.
5.1.1 Linked List

The linked list is a pointer-chasing data structure, where each node holds a next node pointer to the succeeding node. Finding an item in the list entails iteratively chasing through the chain of next node pointers. We are particularly concerned with linked lists in which nodes are sorted according to their integer keys. For the NMP-based linked list, we assume that the entire list is contained in a single NMP vault, and the NMP core combines and sorts the operations that it receives in the order of the requested keys. Sorting the operations allows the NMP core to execute all the combined operations over a single traversal through the list, as shown in Figure 5.1a.

5.1.2 Skiplist

The skiplist is also an ordered, pointer-chasing data structure but with multiple levels of pointers at each node. Each node is assigned a height taken from a particular distribution, and it holds next node pointers to the succeeding node at each level up to its height. All nodes are linked together at level 0 (bottom level), but each node at level $i$ is only half as likely to appear at level $i+1$. Through the multiple levels of pointers and the particular distribution of node heights, the skiplist achieves logarithmic time operation execution, similar to a balanced binary search tree.

The NMP-based skiplist is partitioned across multiple NMP vaults based on pre-defined disjoint ranges of keys, as shown in Figure 5.1b. Each NMP core acts as the combiner for its designated partition, but the multiple NMP cores can execute operations to different parts of the skiplist in parallel.

5.1.3 FIFO Queue

The concurrent FIFO queue is a contended data structure. Unlike with pointer-chasing data structures, queue operations access only the queue head or tail, so they do not incur extensive memory accesses. However, the queue suffers from contention that arises from multiple threads trying to atomically access and modify the same head and tail locations. The NMP-based queue effectively removes contention, for the NMP core becomes the only thread that actually operates on the data structure. It is further optimized by partitioning the queue across multiple NMP vaults, as shown in Figure 5.1c in order to have separate NMP cores running the enq and deq operations.
5.2 Hardware Modification

In my work, the data structures described in Chapter 5.1 were implemented and evaluated on a full-system NMP architecture simulator. Through the evaluation we identified lightweight hardware support that is necessary in order to deliver high performance with NMP.

As mentioned in Chapter 4, we assume that the NMP cores are simple processors without any sophisticated functionality. In particular, we assume that the NMP cores do not have any caches. This is for two reasons. First of all, what can be implemented on the near-memory logic die is restricted by area, power, and thermal constraints [66]. Secondly, computations that are offloaded to NMP are often those that have already been proven to benefit little from cache.

Yet, we observe that data structure operations exhibit temporal and spatial locality at DRAM row granularity. For traversal in linked lists or skiplists, the NMP core reads two words of information from each node, back-to-back: the node’s key value and the pointer to the next node. A single node is stored in contiguous memory (i.e., in the same DRAM row). This implies that data in a single row is accessed twice consecutively. The FIFO queue has an even higher rate of row hits. Queue items are stored successively in memory, according to queued order, so consecutive deq operations access all items in a DRAM row before moving onto the next row.

However, if the NMP core is a simple in-order processor without a data cache, as in our implementation, it can request for only one word of data from memory at a time. Because of this constraint, the NMP vault’s memory controller receives only one data request at a time, and each request gets translated into a separate DRAM access operation, regardless of the row locality among the consecutive requests. Every DRAM access operation takes the costly steps of (1) activating the specified DRAM row, (2) selecting the corresponding columns, and (3) moving data from selected columns in DRAM to the memory controller. The described data access patterns of the data structure operations will result in unnecessary repetitions of these steps. Moreover, data is transferred from DRAM arrays to the memory controller in units of bursts (consecutive columns of data), which is often larger than one word, but because the NMP core requests for only one word of data, the unused extra bits are discarded, when oftentimes the next request will need data from the very same burst.

To address these issues, a small data buffer is added to the NMP vault’s controller. This buffer can be thought of as a single-block cache placed in the memory controller. For the linked list and skiplist, the buffer only needs to be the size of a list node, but for the FIFO queue, it could be as large as an entire DRAM row. The buffer is more advantageous than employing an open-page row-buffer-management policy, which leaves an activated DRAM row open until the row is refreshed or another data request requires a different row to be activated, for two reasons. First, the buffer reduces the column selection delays and redundant burst movement by retaining all bits that arrive with a burst. Secondly, after filling up the buffer, the row can be closed and precharged for the next DRAM operation, which not only hides the precharge delay but also reduces the extra background power drawn by activated rows.

Figure 5.2 shows the added hardware support and the data access process in the modified design.
Only two new hardware components are needed for the proposed hardware support: the buffer itself, which holds the most recently accessed block of data, and a tag register, which holds the tag portion of the buffered block’s memory address. Upon receiving a data read request (step 1), the NMP controller first checks if the tag of the requested data address matches the tag register (step 2). If so, the request is responded to immediately (step 5). This eliminates the DRAM data access process entirely, which significantly reduces data access delays and energy consumption.

If requested data is not in the data buffer, the NMP controller creates DRAM access operations to fill the buffer (steps 3 and 4). If multiple DRAM accesses are necessary, the DRAM access operation that retrieves data for the original request is issued first. This ensures that the data buffer mechanism does not increase the delay in responding to the original data request, while it also allows effective latency hiding for the extra DRAM accesses needed for filling the buffer.

Bits in the tag register are used as valid and ready flags. Once the first DRAM access operation returns with its portion of data, the tag register is updated with the new address, and the buffer is marked valid. However, the ready flag is not set until the buffer has been filled with the corresponding data block. When a memory access request with an address tag that matches the tag register arrives while the buffer is being updated, the request is blocked until the buffer is set to ready. If a write request to the NMP controller modifies data in the buffer, the buffer is simply marked as invalid.

The NMP-based linked list, skiplist, and FIFO queue result in buffer hit rates of 50%, 40%, and 98%, respectively. This inversely shows that it is not necessary to have a fully-fledged data cache in the NMP core, particularly for efficient NMP-based data structures; a small buffer in the memory controller, that takes up much less area and power, is sufficient.

5.3 Evaluation & Analysis

5.3.1 Methodology

We implemented each of the NMP-based data structures outlined in Chapter 5.1 and evaluated them on Brown-SMCSim (described in Chapter 4.2). In particular, we evaluated the NMP-based data structures on architecture with and without the hardware modification described in Chapter 5.2 in order to show the impact of the lightweight hardware modifications. The results of the NMP-based
Table 5.1: Evaluation framework configuration.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>Host cores</td>
<td>8 in-order processors (ARMv7 Cortex-A15), 1 thread/core</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32kB icache, 64kB dcache, private, 2-way set-associative, 2-cycle access latency, 256B/block</td>
</tr>
<tr>
<td>L2 cache</td>
<td>2MB, shared, 8-way set associative, 20-cycle access latency, 256B/block</td>
</tr>
<tr>
<td><strong>NMP configuration</strong></td>
<td></td>
</tr>
<tr>
<td>NMP cores</td>
<td>1 in-order processor/vault (ARMv7 Cortex-A15)</td>
</tr>
<tr>
<td>scratchpad memory</td>
<td>40kB/NMP core, 8kB reserved for host memory-map, stores instructions and program stack.</td>
</tr>
<tr>
<td><strong>Memory Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>vaults</td>
<td>16 vaults, 128MB/vault (total 2GB), 8 DRAM banks/vault</td>
</tr>
<tr>
<td>DRAM parameters</td>
<td>for host-only baseline: 16 main memory vaults</td>
</tr>
<tr>
<td></td>
<td>for NMP: 8 main memory vaults, 8 NMP vaults</td>
</tr>
<tr>
<td></td>
<td>row buffer size: 256B, burst size: 32B</td>
</tr>
<tr>
<td></td>
<td>( t_{RP} ): 13.75,ns, ( t_{RCD} ): 13.75,ns, ( t_{CL} ): 13.75,ns, ( t_{BURST} ): 3.2,ns</td>
</tr>
</tbody>
</table>

Data structures were compared against (1) host-based flat-combining data structures, as a baseline for how the NMP-based algorithm would perform in a non-NMP context, and (2) host-based state-of-the-art concurrent data structures.

Simple benchmarks were used to evaluate the concurrent data structures. The data structure was first populated with random items according to a predefined initial size. A fixed number of random data structure operations were divided equally among a specified number of concurrent host threads, and performance was measured in terms of operation throughput, which refers to the total number of data structure operations completed across all threads in a given period of time.

We ran the simple benchmarks on Brown-SMCSim (described in Chapter 4). Table 5.1 summarizes the configuration details of the evaluation framework.

5.3.2 Linked List Analysis

We implemented the NMP-based linked list (described in Section 5.1.1) on NMP architecture with and without the hardware modification proposed in Section 5.2 (results with the hardware modification are labeled as \textit{NMP ctrl buffer}). For evaluations on unmodified NMP architecture, we gathered results using both open-page and close-page row-buffer-management policies (\textit{NMP open-page} and \textit{NMP close-page}, respectively).

For host-only references, we implemented the flat-combining and operation-sorting linked list (\textit{host FC w/ sort}) [46] and the lazy-lock linked list (\textit{host lazy-lock}) [45]. \textit{Host FC w/ sort} is algorithmically the host-based equivalent of the NMP linked list. \textit{Host lazy-lock} is a state-of-the-art concurrent linked list algorithm; in this algorithm, locking is required only on nodes that are affected by \texttt{add} or \texttt{remove} operations, so in most cases list traversal is executed completely in parallel, unaffected by locking.

For all experiments, we set the initial linked list to be approximately 5MB in total size (2.5 \times the L2 cache size). We set 90% of the random operations to be read-only, and the remaining 10% of the random operations were divided equally among \texttt{add} and \texttt{remove} operations in order to maintain a
constant list size. As noted in Section 5.1.1, only a single NMP vault was used for the NMP-based linked lists.

Figure 5.3 shows the operation throughput for each linked list implementation across varying numbers of concurrent threads. In general, the NMP-based implementations perform better than the host-based implementations, although NMP close-page barely does better than host lazy-lock, with only 2.5% improvement in throughput at eight concurrent threads. However, NMP open-page shows significant improvement and NMP ctrl buffer even more so. At eight concurrent threads, NMP open-page and NMP ctrl buffer each have 16.4% and 73.7% higher operation throughput than host lazy-lock. Note also that the operation throughput of NMP ctrl buffer scales better than that of all other implementations with increasing number of threads.

NMP open-page has 13.5% higher throughput than NMP close-page, and NMP ctrl buffer has 49.3% higher throughput than NMP open-page. This performance difference among NMP-based implementations come from delays associated with DRAM access. In an NMP-based linked list, accessing a node in list traversal generates the following DRAM delays:

1. $t_{RCD}$ to activate the row containing the node’s key,
2. $t_{CL}$ to read the column that contains the node’s key, and
3. $t_{BURST}$ to move the burst containing the node’s key to the NMP controller.

After reading the node’s key value, the NMP core also needs to read the node’s next node pointer. This next node pointer is stored in the same row – better yet, in the same burst – as the node’s key. However, with NMP close-page, the row activated in step (1) is precharged immediately after providing the node’s key, so reading the node’s next node pointer requires repeating steps (1), (2), and (3). On the other hand, NMP open-page leaves the row activated, so reading the next node pointer requires only steps (2) and (3). NMP ctrl buffer goes a step further and caches the burst...
from step (3), so no additional DRAM accesses are needed. Therefore, DRAM delays associated with a single node traversal sum up to:

\[
\begin{align*}
(NMP \ \text{close-page delay}) &= 2 \times (t_{RCD} + t_{CL} + t_{BURST}) \\
(NMP \ \text{open-page delay}) &= t_{RCD} + 2 \times t_{CL} + 2 \times t_{BURST} \\
(NMP \ \text{ctrl buffer delay}) &= t_{RCD} + t_{CL} + t_{BURST}
\end{align*}
\]

To get a better idea of how performance is affected by the various NMP and host schemes, we also looked into the memory accesses of each linked list implementation. Figure 5.3b shows the number of memory read requests that arrive at memory controllers, the number of DRAM read accesses (read bursts) that the read requests are translated to, and the number of DRAM row activations that happen in order to serve these bursts. In the NMP-based implementations, the number of DRAM bursts are at most equal to the number of memory requests, for each request asks for only one word of data. Although all NMP-based implementations make the same number of read accesses to the memory controller, the number of DRAM row activations is reduced to half with NMP open-page, and the number of DRAM read bursts is reduced to half with NMP ctrl buffer.

In host-based implementations, each memory read request to the memory controller originates from a cache miss; therefore it requests for a cache block size chunk of data. A cache block size is normally a multiple of the DRAM burst size, so the number of DRAM read bursts is much larger than the number of read requests. (In our architecture framework, the cache block size was equal to a DRAM row size and was eight times the burst size.) Because of the sheer number of DRAM read bursts, host-based linked lists perform worse than the NMP-based linked lists. However, although host lazy-lock has much more read bursts than host FC w/ sort, host lazy-lock shows higher throughput since the memory requests and DRAM bursts are issued and processed in parallel, whereas the memory requests in host FC w/ sort are issued sequentially by a single combiner thread.

5.3.3 Skiplist Analysis

We implemented the NMP-based skiplist as defined in Section 5.1.2 with 1, 2, 4, and 8 NMP vaults (NMP 1part, 2part, 4part, 8part). We also implemented the host-based partitioned and flat-combining skiplists (host FC 1part, 2part, 4part, 8part) as references on how the NMP-based algorithm would perform in a host-only setting. As described in Section 5.1.2, partitioning achieves higher concurrency by allowing multiple NMP cores (or in the case of host FC skiplists, multiple combiner threads) to safely operate on different parts of the skiplist in parallel.

Just as we did in evaluating the NMP-based linked list, NMP-based skiplists were evaluated both with and without the memory controller modifications described in Section 5.2. For evaluations without the memory controller modifications, we used the open-page row-buffer-management policy for NMP vaults. We refer to the NMP-based implementations with and without the hardware modifications as NMP ctrl buffer and NMP open-page.
Figure 5.4: Baseline NMP-based skiplist evaluations. In 5.4a, solid lines and dashed lines show the operation throughput of NMP-based implementations with and without hardware modifications, respectively. Dotted lines show the operation throughput of host-based implementations. 5.4b shows the memory access and DRAM activity analysis for each skiplist implementation, based on an 8-thread execution.

We also implemented the state-of-the-art host-based lock-free skiplist (host LF) [34] for comparison. Host LF allows complete concurrency among host threads by using only atomic compare-and-swap operations to apply changes to the skiplist.

For a baseline skiplist evaluation, we initialized each skiplist to be approximately 12MB in size (6× larger than the L2 cache size). We set 90% of the random operations to be read-only and divided the remaining 10% of the operations equally among add and remove operations, just as we did in the linked list evaluations.

Figure 5.4a shows the operation throughput across varying number of concurrent threads, with different skiplist implementations. Results from 1 and 2 partition skiplists were omitted from the Figure in order to highlight the most relevant and interesting results.

Just as it had been in the linked list case, adding a node-size data buffer to the NMP vault’s memory controller improves the NMP-based skiplist performance. This is because node traversal in the skiplist is subject to DRAM access delays similar to the linked list’s node traversal: (1) $t_{RCD}$ to activate the row containing the node, (2) $t_{CL}$ to read the column containing the node’s key, and (3) $t_{BURST}$ to move the burst containing the key. After reading the key, the node’s next node pointer is accessed. The next node pointer access does not cause any additional DRAM access delay with NMP ctrl buffer, whereas an additional ($t_{CL} + t_{BURST}$) delay is generated with NMP open-page. Although the memory access and DRAM activity analysis of the various skiplist implementations
in Figure 5.4b show that *NMP ctrl buffer* implementations generate much more DRAM read bursts than *NMP open-page* implementations\(^1\) the latency-hiding mechanism described in Section 5.2 allows *NMP ctrl buffer* to still yield higher performance.

However, the improvement in skiplist performance provided by *NMP ctrl buffer* is modest compared to the improvement in linked list performance. This is due to the data traversal patterns particular to the skiplist. Every skiplist operation entails traversing through data structure: for a `contains` operation, to find the node in question, and for an `add` or `remove` operation, to find the specific location to apply the operation on. The skiplist is inherently a balanced tree-like data structure, so the traversal begins at the few top-level nodes and moves down to lower levels using binary search. In order to move down a level in this binary search, the traversal needs to backtrack to the previous node at the current level, but because the buffer has already been overwritten with the current node, the previous node has to be accessed from DRAM once again. This results in a lower buffer hit rate (as shown in Figure 5.4b) compared to the linked list, and it limits the performance improvement provided by the buffer placed in the memory controller.

What is also notable in the baseline evaluation is that *host LF* completely outperforms the NMP-based implementations, contrary to the projection in [77]. We find that cache effects account for the relatively high performance of host-based skiplists. In fact, Figure 5.4b shows that the host-based implementations have significantly less memory read requests to the DRAM compared to the NMP-based implementations. This, again, is due to the skiplist’s hierarchical structure. Because every operation on the skiplist begins traversal at the few higher-level nodes, these few higher-level nodes are accessed repeatedly over the course of many operations. As a result, in a non-NMP context, these nodes are likely to remain in cache, and only a few node accesses per operation actually go out to memory.

We did a sweep on cache sizes to find the skiplist size relative to the cache size that makes using

\(^1\)Because a skiplist node contains multiple levels of pointers, multiple DRAM bursts are needed in order to fill the memory controller buffer with a full skiplist node.
the 8-partition NMP-based skiplist more advantageous than using the host lock-free skiplist. In these experiments, we set the initial skiplist size to approximately 850MB. The breakdown of the random operations were kept the same as before.

Figure 5.5 shows the operation throughput of NMP-based and lock-free skiplists, with varying ratios of last-level cache size to skiplist size. As expected, the host lock-free skiplist’s performance is heavily influenced by its size relative to cache, whereas the NMP-based skiplist’s performance is insensitive to cache size.

5.3.4 FIFO Queue Analysis

Queues are inherently different from the pointer-chasing data structures we considered earlier. Items are usually stored in contiguous memory, so they do not suffer from the same memory access delays. Nevertheless, they do suffer from contention among multiple threads trying to access and modify the head and tail of the data structure.

We implemented the NMP-based queue as described in Section 5.1.3 on NMP architecture with and without the NMP controller data buffer (NMP ctrl buffer and NMP open-page, respectively). The NMP core effectively removes contention, for it is the only thread that operates on the data structure. As the equivalent host-based algorithm, we also implemented the host-based dual flat-combining queue (host dual FC), which has separate combiner threads for enq and deq operations. For the state-of-the-art concurrent queue, we implemented the host-based lock-free queue (host LF) [82], in which concurrent threads use atomic compare-and-swap operations to safely execute enq and deq operations.

There are two ways to implement a queue. One is implementing it as a circular array buffer with indices specifying the head and tail (array-based), and another is implementing it as a linked-list with pointers to the head and tail (list-based). The array-based implementation uses fewer atomic operations for each enq or deq operation compared to the list-based implementation, but it requires setting aside a fixed-size block of memory to store the queue items. We considered both implementations for the host-based queues. However, the NMP-based queues were implemented array-based only, for the NMP vault is already memory dedicated to store the data structure.

For evaluation, we set the initial queue size to be 2MB (4MB for the list-based implementations because each queue item needs to store a pointer to the next item, which is unnecessary in the array-based implementations). Random operations on the queue were divided as 50% enq and 50% deq.

Figure 5.6a shows the operation throughput of various queue implementations across varying number of concurrent threads. At eight concurrent threads, NMP open-page and NMP ctrl buffer each yield 3.5× and 3.7× higher throughput than the array-based host dual FC, just by removing contention. Compared to the list-based host LF, NMP open-page and NMP ctrl buffer show 40.5%

In reality, the skiplist size could be much larger than the size that we experiment with, but we were restricted by memory address space limitations in the architecture framework to further increase the skiplist size. We therefore simulated the effects of having a larger data structure by reducing the cache size.
Figure 5.6: NMP-based FIFO queue evaluations. In 5.6a, solid lines show the operation throughput of NMP-based implementations, dashed lines show that of array-based host implementations, and dotted lines show that of list-based host implementations. 5.6b shows the memory access and DRAM activity analysis of NMP-based queues at 8 concurrent threads.

and 48% higher throughput at eight concurrent threads.

As mentioned earlier, the array-based host LF does fewer atomic operations for each queue operation than the list-based host LF and thus yields higher operation throughput than the NMP-based queues. However, while the operation throughput for host-based queues flatten out due to contention, the throughput for NMP-based queues scale linearly, and we expect the NMP-based queues to outperform even the array-based host LF with more concurrent threads.

At eight concurrent threads, NMP ctrl buffer has 5.3% higher operation throughput compared to NMP open-page. Figure 5.6b shows the number of memory read requests, DRAM read bursts, and DRAM row activations for the two NMP-based queues, based on an 8-thread execution. Among all the data structures that we considered, the queue has the highest buffer hit rate (98.3%, while the linked list and skiplist had approximately 50% and 40% hit rates, respectively), and this greatly reduces the number of DRAM bursts and row activations.

However, the impact of the data buffer on overall performance is not as dramatic compared to the pointer-chasing data structures because of Amdahl’s law. In pointer-chasing data structures, successive memory accesses consume a large portion of the execution time in a data structure operation. On the other hand, each queue operation requires only one memory access, and because this is such a small portion of the total operation execution time, reducing the memory access latency improves the overall performance only slightly.
Chapter 6

NMP-Enabled Cache-Conscious Hybrid Data Structures

In the first part of my dissertation work (Chapter 5), the performance of NMP-based concurrent data structures was verified on a cycle-accurate, full-system NMP architecture simulator. Through the process, hardware effects in the traditional memory hierarchy that impacted conventional data structure performance were identified. While Liu et al.’s NMP-based data structures [77] were carefully designed to preserve high concurrency, they overlooked two things: (1) data access from NMP cores are still subject to delays that arise from DRAM activity, which make up a large portion of memory access delay, and (2) even a large pointer-chasing data structure can have portions with high locality, and the benefits of utilizing on-chip cache are lost when such data structures are naively offloaded entirely to NMP.

The second part of my dissertation work builds up on these observations and focuses on further enhancing latency-sensitive data structures that have been heavily optimized to exploit on-chip cache locality of conventional systems. We make the observation that such data structures are often pointer-chasing yet hierarchical data structures, in which nodes are organized by levels in a top-down hierarchy. Importantly, the higher-level nodes exhibit better cache locality than lower-level nodes, for traversals on these data structures always begin at the few top-level nodes but diverge to many different node paths while moving down the hierarchy. For this reason, the access frequency of each node depends heavily on its position in the hierarchy.

Based on this observation, we propose and empirically evaluate cache-conscious hybrid data structures enabled by NMP architecture (we also called these NMP-hybrid data structures). The NMP-enabled hybrid data structures effectively combine host-based and NMP-based data structure designs to retain and further enhance cache-locality benefits of hierarchical data structures.
6.1 Hybrid Data Structure Design

6.1.1 Overview

We focus on two representative hierarchical data structures: (i) skiplists [95] and (ii) B+ trees [11, 27]. Both of these data structures are hierarchical, meaning that nodes comprising the data structure are organized by levels in a top-down hierarchy. The topmost level consists of either one node (B+ tree) or very few nodes (skiplist), and the number of nodes at each level below is a multiple of the number of nodes at the level above. Lookups on these data structures always begin at the topmost level, but they diverge to different node paths as the lookups move down the hierarchy. For this reason, the access frequency of each node in the structure depends heavily on its position in the hierarchy; that is, nodes in the higher levels are accessed much more frequently (and thereby benefit more from caching) than nodes in the lower levels.

The key insight of our NMP-enabled hybrid data structures is to leverage the skew in access frequency that stems from the data structure’s topology. Essentially, we divide each data structure into a host-managed portion consisting of higher levels of the data structure and an NMP-managed portion consisting of lower levels. The host-NMP split point (i.e., the level at which to divide the data structure) is chosen such that the entire host-managed portion can fit within the last-level cache, effectively “pinning” the higher-level nodes without any additional hardware modifications. The remaining lower levels of the data structure are placed in NMP-capable memory, accessed and managed entirely by NMP cores, which prevents the infrequently accessed lower level nodes from polluting the host cache.

Despite the seemingly simple concept of our idea, rigorous technical detail at the algorithmic level to ensure high concurrency and correctness in these hybrid data structures. Conventional concurrent data structures have been designed under the assumption that an operation will be applied onto a single coherent structure, but a hybrid data structure is divided into two parts, each accessed and managed by different sets of threads (i.e., host cores vs. NMP cores). The data structure as a whole must nevertheless be kept coherent throughout the entire operation execution. This prompts new synchronization problems. Careful coordination among the host threads, among the NMP cores, and between host and NMP cores is essential. Moreover, while skiplists and B+ trees have similar hierarchical topologies, the two data structures by design have very different correctness conditions for concurrent modifications, so each of the hybrid algorithms must be designed accordingly.

For concurrency in the NMP-managed portion, we build upon prior work that applies flat-combining [46] to NMP-based data structures [23, 77]. We first elaborate on the implementation of NMP-managed portions of the hybrid data structures (Chapter 6.1.2). The following subsections then provide details on the hybrid skiplist and hybrid B+ tree designs (Chapters 6.1.3 and 6.1.4 respectively). Lastly, Chapter 6.1.5 describes an optimization that further improves the performance of hybrid data structures.

While this work provides complete hybrid implementations for only the skiplist and B+ tree,
note that the design pattern for NMP-enabled hybrid data structures are applicable to any cache-conscious hierarchical data structure.

### 6.1.2 NMP-Managed Portion of Hybrid Data Structures

As described in Chapter 5.1, conventional flat-combining data structures have a single designated combiner thread that actually applies operations to the data structure. Each concurrent thread simply posts its operation request to an assigned slot in the publication list, which the combiner thread iterates through to check for unserved operation requests. In NMP-based flat-combining data structures, each NMP core becomes the combiner thread for the portion of the data structure stored in its coupled NMP partition.

As described in Chapter 4, we assume that each NMP core is equipped with a small scratchpad memory. A portion of the scratchpad memory is memory-mapped into the host address space so that it can be used as the publication list. Note that our evaluation framework for this work, Brown-SMCSim \[22\] (Chapter 4.2), accurately models latencies caused by reads and writes on the memory-mapped scratchpad memory of NMP cores.

In order to offload a data structure operation to the NMP-managed portion, the host thread writes the following to its assigned slot in the target NMP core’s publication list:

1. lookup key (4 bytes),
2. associated value for update or insert operations (4 bytes),
3. pointer to the node from which the NMP core should begin its traversal, which we refer to as begin-NMP-traversal node (4 bytes),
4. operation type – read, update, insert, remove, or other operations for maintenance and debugging (3 bits), and
5. valid-bit flag indicating that the slot contains a valid operation (1 bit).

After offloading an operation, the host thread polls on the valid-bit to check the operation’s status.

In non-hybrid NMP-based data structures \[23\ 77\], item 3, begin-NMP-traversal node, had been unnecessary because the NMP core traversed the entire data structure, and therefore all traversals began at a fixed sentinel node within the NMP partition. However, in hybrid data structures, host-side traversals become shortcuts into NMP-managed lower levels, and NMP-side traversals can begin at one of many nodes referenced from the bottommost level of the host-managed portion. (Although NMP-side nodes cannot be accessed directly by a host thread, the host-managed portion keeps references to node pointers in the NMP-managed region for this reason.)

However, because the NMP core processes operations in its publication list one at a time, the begin-NMP-traversal node of an operation may end up being modified by a concurrent operation that gets processed earlier in the NMP core. This is a new synchronization problem that affects correctness in hybrid data structures, and therefore the hybrid skiplist and the hybrid B+ tree
provide tailored mechanisms for the NMP core to detect such issues, in which case the NMP core aborts the operation and notifies the host thread to retry the operation accordingly.

When the NMP core completes processing an offloaded operation, it writes the following to the publication list before clearing the valid bit:

1. retry flag if the NMP-side operation requires a retry from the very beginning (1 bit),
2. return value indicating the operation's success or failure (1 bit),
3. associated value for read operations (4 bytes), and
4. NMP-side’s new node pointer for successful insert operations (4 bytes).

Flat-combining handles the correct application of concurrent operations to the NMP-managed portion of an NMP-hybrid data structure. In order to further increase parallelism, the NMP-managed portion of the data structure is divided across across multiple NMP partitions, similar to the NMP-based skiplist described in Chapter 5.1.2. Because each NMP partition is an isolated portion of memory accessed and modified exclusively by its coupled NMP core, the partitioning allows operations to different partitions to run in parallel without data races or coherence issues.

6.1.3 Hybrid Skiplist

The skiplist is an ordered, pointer-chasing data structure with multiple levels of pointers at each node. Each node is assigned a height, and a node holds next-node pointers to the succeeding node at each level up to its height. The height is taken from a particular distribution such that all nodes are linked together at level 0 (bottommost level), but each node at level \( i \) is only half as likely to appear at level \( i + 1 \). The skiplist is usually configured with \( \log_2 N \) total levels, where \( N \) is the number of key-value pairs at initialization.

A lookup on the skiplist is a top-down search through the levels. The lookup initially traverses through next-node pointers at the topmost level, but once it reaches a node with a key larger than the lookup key, it backtracks to the previous node on the same level and continues the traversal at the level immediately below. This process is repeated until the lookup key is found. The particular distribution of node heights allows for a logarithmic time execution, similar to a balanced binary search tree.

Figure 6.1 depicts the hybrid skiplist design. The cumulative size of the top \( x \) levels of a skiplist can be estimated as \( 2^x \text{sizeof(Node)} \). The number of levels to be placed in the host-managed region is chosen such that this is approximately equal to the last-level cache size.

The host-managed portion of the hybrid skiplist is implemented according to the state-of-the-art lock-free skiplist \[36, 47\]. Nodes in the host-managed portion hold references to their counterparts in the NMP-managed portion, in order to provide shortcuts (i.e., reference to begin-NMP-traversal node) for NMP-side traversals. The NMP-managed portion is implemented according to the NMP-based flat-combining data structures described in Chapter 6.1.2. Here, we simply assume that nodes in the NMP-managed portion are distributed across NMP partitions based on pre-defined, equal-size ranges of keys. The data structure stores information on the key ranges so that the target NMP
partition for an operation can easily be determined with the lookup key. The algorithm can be extended to other partitioning schemes for reasons such as better load balancing, but we leave this to future work.

Guaranteeing correctness in the hybrid skiplist is relatively straightforward. For correctness, the skiplist data structure simply needs to always maintain the skiplist property \( \text{set of nodes at every level } i \text{ is a subset of all nodes at level } i - 1 \). The lock-free skiplist and the NMP-based flat-combining skiplist both correctly maintain these properties by applying insertions from bottom to top and by applying removals from top to bottom. In the hybrid skiplist, we ensure the same by applying insertions in the NMP-managed portion first, and then in the host-managed portion; likewise, removals are applied in the host-managed portion first, and then in the NMP-managed portion.

Data modifications in the NMP-managed portion of the data structure that may affect correctness are cases where the begin-NMP-traversal node of an NMP-offloaded operation is removed by an earlier-processed concurrent operation. To ensure that such modifications are always detected before the NMP-side traversal begins, a remove target node is always logically marked as deleted, even though the NMP core immediately processes the physical removal as well.

Listing 6.1 shows pseudocode for the insert operation on the host-side. After the initial traversal through the host-side (line 7), the begin-NMP-traversal node for the operation is identified (lines 15-16). Then, the SEND_NMP_INSERT_OPERATION (line 17) writes operation information to the thread’s assigned slot in the target NMP partition’s publication list, according to the description in Chapter 6.1.2. Note that most of the steps in Listing 6.1 apply to read, update, and remove operations as well.

Once the operation is offloaded to a target NMP core, the NMP core carries out its portion of the operation as described in Listing 6.2. Line 7 checks if the begin-NMP-traversal node has been marked as deleted by an earlier-processed concurrent operation, in which case the retry flag is set for the host thread (lines 8-9). The remainder of the operation is carried out according to the single-threaded skiplist algorithm (lines 14-24). Note that the described process is identical across all skiplist operations.

The hybrid skiplist retains linearizability [51], which is a strong correctness guarantee for concurrent data structures. Linearizable data structures are identified by linearization points, which
bool insert(int key, int value, int thread_id) {
    Node* preds[HOST_HEIGHT];
    Node* succs[HOST_HEIGHT];
    Node* newnode, foundnode, nmpnode = NULL;
    int part_id, height = 0;
    bool ret = false;
    foundnode = find(key, head, preds, succs); // finds position of node associated with key
    and fills in preds, succs arrays with pointers to the predecessor and successor nodes at
    each level
    if (foundnode != NULL) { return false; }
    part_id = get_partition(key);
    height = rand_height(); // height of new node
    if (height > NMP_HEIGHT) {
        newnode = new Node(key, value, height-NMP_HEIGHT);
        newnode->nmp_ptr = NULL;
    }
    if (part_id == get_partition(preds[0]->key)) {
        nmpnode = preds[0]->nmp_ptr;
    }
    SEND_NMP_INSERT_OP(); // according to Ch 6.1.2
    if (GET_NMP_RETRY_FLAG()) {
        if (newnode != NULL) { delete newnode; }
        // retry from beginning
    } else {
        ret = GET_NMP_RETURN_VALUE();
        if (newnode != NULL) {
            if (!ret) {
                delete newnode;
                return false; }
            newnode->nmp_ptr = GET_NMP_NODE_PTR();
            // From here, link newnode into host-side levels according to LF skiplist algorithm
            // (code omitted)
            publist[i].nmp_ptr = newnode;
            publist[i].return_value = 1; }
    } return ret; }

Listing 6.1: Hybrid skiplist: host-side insert operation.

Node* preds[NMP_HEIGHT];
Node* succs[NMP_HEIGHT];
Node* curr, foundnode, newnode;
// publist[i] contains operation information offloaded from thread i
if (publist[i].nmp_ptr != NULL) {
    curr = publist[i].nmp_ptr;
    if (MARKED(curr)) {
        // curr has been removed by a concurrent remove.
        // set RETRY flag and move on to next op (code omitted)
    } else {
        foundnode = find(key, curr, preds, succs); // finds position of node associated with key
        if (foundnode != NULL) {
            publist[i].return_value = 0;
        } else {
            newnode = new Node(publist[i].key, publist[i].value);
            newnode->height = NMP_HEIGHT;
        }
        newnode->nmp_ptr = publist[i].nmp_ptr;
        newnode->host_ptr = publist[i].host_ptr;
        // link newnode into skiplist according to single-threaded skiplist algorithm (code omitted)
        publist[i].nmp_ptr = newnode;
        publist[i].return_value = 1; }

Listing 6.2: Hybrid skiplist: NMP-side insert operation.
are single points at which operations take instantaneous effect. The pseudocode in Listings 6.1 and 6.2 shows that an insertion in the hybrid skiplist takes instantaneous effect when the new node is linked into the NMP-managed portion. Although we omit pseudocode for other operations, the remove operation also takes instantaneous effect when the removal target node is removed from the NMP-managed portion.

The linearization points for successful read and update operations are points where the value associated with the target node is successfully read or updated, whether in the host-managed or NMP-managed portion. However, in order to account for concurrent insert and update operations on the same key (where the update is offloaded to NMP after the new node is linked in the NMP-managed portion but before it is linked in the host-side levels) the NMP-side update returns the target node’s host.ptr through the publication list. Using this information, the host-side update ensures that the new value is updated in the host-side node as well. This process guarantees future read operations to read correctly updated values.

6.1.4 Hybrid B+ tree

The B+ tree is an n-ary tree data structure consisting of a root, inner nodes, and leaf nodes. Leaf nodes store up to n key-value pairs in sorted order; non-leaf nodes (i.e., the root and inner nodes) hold up to n child node pointers to subtrees, along with dividing keys. A subtree to the left of a dividing key must only contain keys that are less than or equal to the dividing key; similarly, a subtree to the right of a dividing key must only contain keys that are greater than the dividing key. Lookups on a B+ tree require top-down pointer-chasing from the root to the leaf containing the lookup key, the traversal path guided by dividing keys in the non-leaf nodes.

The B+ tree also maintains the following invariants: (i) each non-root inner node has at least ⌈n/2⌉ children, (ii) each leaf node holds at least ⌈n/2⌉ items, and (iii) the path from the root to any leaf node is the same length. Note that the value of n in in-memory OLTP systems is chosen such that the B+ tree nodes align with the cache block size (typically 64 or 128 bytes), which is to fully exploit spatial locality [97]. The resulting n is often fairly large—for example, n is 16 for the B+ tree used in the DBx1000 framework [121].

To simplify the hybrid B+ tree’s configuration phase, we assume that the initial B+ tree is constructed over an existing database table. This is not uncommon when building index structures for in-memory OLTP systems, which is a major use case of B+ tree structures. Furthermore, for a sufficiently large B+ tree, the hybrid B+ tree does not require frequent reconfiguration: the initial setup becomes inappropriate only when the host-managed portion becomes significantly larger than the last-level cache, which requires the B+ tree to grow by several magnitudes in size[6]. We assume that enough memory (including the NMP partitions) is provisioned to account for the data structures and their expected growth.

1 In such cases, the hybrid B+ tree will indeed need to be reconfigured. Because this would be an infrequent operation, we expect that it could be done during a maintenance phase, when no other concurrent operations are made on the B+ tree. However, we leave this to future work.
Figure 6.2: Hybrid B+ tree design.

Figure 6.2 illustrates the hybrid B+ tree design. The hybrid B+ tree is first constructed entirely in the host-managed region. The host-NMP split point for the hybrid B+ tree is determined based on the cumulative size of the top $x$ levels. This is approximately $(1 + r \sum_{i=0}^{x-2} m^i)\text{sizeof(Node)}$, where $r$ is the number of children from the root node and $m$ is the average fanout of the non-root inner nodes. $m$ ranges between $\frac{1}{2}n$ and $\frac{2}{3}n$, depending on the order in which initial keys are populated into the tree \[104\].

The lower levels for the NMP-managed portion are divided into NMP partitions at boundaries chosen based on the root’s grandchildren. For example, if the root has exactly $rm$ grandchildren and the system has $p$ NMP partitions, lower levels under the first $\frac{rm}{p}$ grandchildren nodes are pushed down to the first partition, and lower levels under the second $\frac{rm}{p}$ grandchildren nodes are pushed down to the second partition. Pointers from the bottommost level nodes of the host-managed portion are references to child nodes stored in the NMP partitions. The NMP partition information is stored with the pointer reference: since B+ tree nodes are aligned at 64 or 128 bytes, we exploit unused least significant bits of the NMP-side node pointer to store the corresponding NMP partition’s ID. The initialization phase completes once all lower levels constituting the NMP-managed portion are pushed down to appropriate NMP partitions.

Once the initialization phase is complete, synchronizing concurrent operations becomes the challenge in hybrid B+ trees. In general, providing concurrency in B+ trees is nontrivial, for insertions may trigger structural changes across multiple levels to satisfy B+ tree invariants. Although infrequent, structural changes in upper levels are driven by changes in lower levels, and changes at all affected levels must be completed in order to bring the B+ tree to a correct state. This is a major difference from the skiplist, where the linking of an inserted node at each level is a stand-alone operation that does not interfere with the correctness of the overall structure. While the same applies to deletions, we relax the B+ tree invariant on the minimum number of items in the leaf node to simplify synchronization with regards to remove operations, which is an approach also taken in prior work \[60, 85, 102, 123\]. Even so, the hybrid B+ tree requires a more complicated synchronization method than the hybrid skiplist.

Furthermore, the hybrid B+ tree must be capable of synchronizing operations with regards to modifications in the begin-NMP-traversal node. For example, in the hybrid B+ tree of Figure 6.2 an insert(key=57) operation will split nodes D and C, and in turn, node B will be updated to hold a reference to node C’ that has been split from C. However, a concurrently-offloaded operation may
Listing 6.3: Hybrid B+ tree: node definitions.

```c
// host-managed portion:
struct InnerNode {
    volatile int seqnum;
    int level;
    int slotuse; // # of slots in use
    int dividing_keys[SLOTMAX];
    Node *children[SLOTMAX+1];
};

// NMP-managed portion:
struct InnerNode {
    int parent_seqnum; // for synchronization at host-NMP boundary
    int level;
    short lock; // 1 if locked, 0 otherwise
    short slotuse;
    int dividing_keys[SLOTMAX];
    Node *children[SLOTMAX+1];
};
// LeafNodes have the same structure, except they hold data keys and values,
// instead of dividing keys and child ptrs.
```

also have had the original node C as its begin-NMP-traversal node; if this operation gets processed by the NMP core after the `insert(57)`, it may be applied incorrectly, for some subtrees under the original node C cannot be accessed from node C at that point. The NMP-side of the hybrid B+ tree algorithm must be capable of detecting such cases to ensure correctness.

Sequence locks [16] are at the core of our hybrid B+ tree algorithm. The sequence lock is essentially a sequence number that is atomically incremented at the beginning and end of a critical section that involves writes. An operation that reads data protected by the sequence lock checks the sequence number at the beginning and end of its operation to verify that the data has remained unchanged in the meanwhile (if the data has changed, the operation can abort and restart). Because atomic operations on the sequence locks are issued only for writes, synchronization traffic is reduced compared to read-write locks. We also use the sequence numbers as version numbers for synchronization across the host-NMP boundary.

Listing 6.3 specifically shows how nodes are defined in our hybrid B+ tree implementation. Each node in the host-managed portion has a sequence number associated with it (line 3). However, because the NMP-managed portion is accessed by a single NMP core only, the sequence lock is unnecessary. Each node in the NMP-managed portion instead has a simple lock (line 13). It also keeps track of its parent node’s latest sequence number (line 11), which is used to manage synchronization issues at the begin-NMP-traversal node.

To explain how these synchronization mechanisms are used to ensure correctness, we describe the `insert` algorithm of the hybrid B+ tree. The pseudocode in Listing 6.4 shows the host-side of the algorithm. As the host thread traverses down the tree, it records all nodes on the traversal path and their sequence numbers at the time of access (lines 6-7, 13-15). If a destination child node is being modified at the time of access, the traversal first waits for the write on the child to complete (lines 12-14). Afterwards, if the current node has remained unchanged, the traversal moves down the tree (lines 15-18); otherwise, the traversal moves back up the tree to the lowest ancestor node that has not been modified during the operation’s execution (iterations of lines 19-23). Upon reaching the last host-side level of the B+ tree, the host thread offloads the `insert` operation to an appropriate
Node* path[TREE_HEIGHT]; int local_seqnum[TREE_HEIGHT];
Node* curr, child = NULL; int curr_level = 0;
bool retry_from_root = true;
while (retry_from_root) {
  curr = root; curr_level = curr->level;
  local_seqnum[curr_level] = curr->seqnum;
  path[curr_level] = curr;
  if (local_seqnum[curr_level] % 2 != 0) continue;
  while (curr_level < TREE_HEIGHT) {
    child = find_child(curr, key);
    if (curr_level > LAST_HOST_LEVEL) {
      do {
        local_seqnum[curr_level-1] = child->seqnum;
      } while (local_seqnum[curr_level-1] % 2 != 0);
      path[curr_level-1] = child;
      if (curr->seqnum == local_seqnum[curr_level]) {
        curr_level--;
        curr = child;
      } else {
        // curr has been modified in the meanwhile,
        // move back up the path
        curr_level++;
        curr = path[curr_level];
      }
    } else {
      // reached last host-side level
      SEND_NMP_INSERT_OP(); // according to Ch 6.1.2
      if (GET_NMP_RETRY_FLAG()) break; // retry from root
      if (GET_NMP_LOCK_PATH()) {
        locked_all = false;
        for (i = LAST_HOST_LEVEL; i < TREE_HEIGHT; i++) {
          if (!CAS(path[i]->seqnum, local_seqnum[i], local_seqnum[i]+1)) {
            // failed to lock node at level i:
            // unlock locked nodes on path (code omitted)
            SEND_NMP_UNLOCK_PATH_OP();
            break; } // retry insert from root
        if (path[i]->slotuse < INNER_SLOTMAX) {
          locked_all = true; break; }
        }
      }
    }
  }
  if (locked_all) {
    SEND_NMP_RESUME_INSERT_OP();
    retry_from_root = false;
    // RESUME_INSERT is guaranteed to succeed.
    // From here, complete host-side insertion process
    // following the single-threaded insert algorithm
    // unlock all nodes on path and return (code omitted)
  }
} else { // of if GET_NMP_LOCK_PATH
  retry_from_root = false;
  // insert completed within NMP partition. return.
  break; }
}

Listing 6.4: Hybrid B+ tree: host-side insert operation.
The appropriate NMP-side child node reference held in the last host-side node becomes the begin-NMP-traversal node.

Listing 6.5 shows pseudocode for the NMP-side insert algorithm. Lines 3-9 deal with synchronization at the host-NMP boundary, but we first describe the insertion process assuming that the begin-NMP-traversal node has not been split by a concurrent operation. In lines 10-12, the NMP core traverses down the tree, recording all nodes along the traversal path. Upon reaching the bottom, the NMP core traverses back up the path to lock each node that will be affected by the insert, starting from the leaf node (lines 14-20). Whether a node will be split is determined by the number of slots in use: if a node is currently full, the insertion will split the node, and in turn its parent node will be affected and must be locked as well. This is iteratively applied until a non-splitting node is reached. If all affected nodes are contained within the NMP-managed portion, the insertion takes place immediately, following the single-threaded B+ tree insert algorithm, and the path is unlocked (lines 26-28). However, if even the topmost level node on the NMP-side path requires a node split, the NMP core notifies the host thread (via the publication list slot) to lock nodes on the host-side path accordingly (lines 29-30). In this case, the NMP-side nodes remain locked, in order to prevent concurrent insert or remove operations from modifying any of the affected nodes in the meanwhile (lines 21-24).

If the host thread receives a LOCK_PATH command from the NMP core, it proceeds to lock (i.e., increment the sequence numbers of) affected nodes on the host-side path, based on the traversal
path and sequence numbers that it recorded during the initial traversal down the tree (Listing 6.4 lines 28-38). Once all affected nodes are locked, the host thread offloads a RESUME_INSERT operation to the NMP core (line 40). However, if the locking fails at any point (line 31), the host thread unlocks all acquired locks on the host side (lines 32-33), notifies the NMP core to unlock the path accordingly (line 34), and then retries the operation from the beginning (line 35).

Once the NMP core receives the RESUME_INSERT operation, the NMP core completes the insertion and unlocks the NMP-side path, for the insertion is guaranteed to succeed at this point. Before notifying the host thread of the operation completion, the NMP core increments the parent sequence number of the topmost level nodes on the operation path (including the newly split-off node) to reflect the eventual unlocked sequence number of their parent nodes. After the NMP-side process completes, the host thread completes its portion of the insertion process and unlocks the affected nodes (i.e., increments their sequence numbers).

The parent sequence number recorded in the begin-NMP-traversal node (hereby referred to as recorded parent#) is used for synchronization at the host-NMP boundary. When a host thread offloads an operation to the NMP core, it writes the sequence number of the last host-side node (i.e., parent of the begin-NMP-traversal node) to the publication list slot, in addition to other operation information mentioned in Chapter 6.1.2. Before the NMP-side tree traversal begins, the NMP core compares the offloaded parent sequence number (referred to as offloaded parent#) against the recorded parent# (Listing 6.5 lines 2-9). If the recorded parent# is greater than the offloaded parent#, this indicates that the host-side parent node has been modified after the operation offload, due to a node split in the begin-NMP-traversal node. This implies that the corresponding leaf node for the operation may have become unreachable from the begin-NMP-traversal node, so the NMP core notifies the host thread to retry this operation. However, there could also be cases where the offloaded parent# is greater than the recorded parent#. This indicates that the host-side parent node had been modified due to a split in a sibling node; in this case, the recorded parent# is simply updated to the offload parent# for consistency.

Synchronizing read, update, and remove operations is relatively straightforward. These operations follow the insert operation’s initial host-side and NMP-side traversal process (Listing 6.4 lines 4-25, Listing 6.5 lines 2-13). Once the NMP core reaches the leaf node, read or update operations can be applied immediately. However, the remove operation cannot be applied if the leaf node is in a locked state, for the removal affects the number of slots in use at the node, which in turn affects the node split process being prepared by a concurrent insertion. In this case, the remove operation is aborted and retried from the beginning.
6.1.5 Optimization: Non-blocking NMP Calls

In our proposed hybrid data structures, once a host thread executing a data structure operation reaches the bottom level of the host-managed portion, it sends the operation request to an appropriate NMP core. These NMP calls are assumed to be blocking operations; that is, the host thread actively waits until the NMP-side of the operation is done. This poorly utilizes compute resources and limits throughput, as shown in Figure 6.3a.

In order to increase concurrency and overall throughput, we modify the NMP calls to be non-blocking. With non-blocking NMP calls, a host thread can offload an operation to an NMP core and immediately move on to process the next pending operation, while the NMP core does its portion of the work (Figure 6.3b). Host threads can maintain a list of ongoing operations to later retrieve results returned by NMP cores.

Data structure operations with non-blocking NMP calls involve only minor changes to the high-level API. In typical data structure APIs, a high-level function call simply returns a Boolean flag indicating the success of an operation. With non-blocking NMP calls, the function call should instead return an operation ID number as an acknowledgement that the operation is now in progress. The API should also be extended with a separate function that takes the operation ID as input in order to check on the operation’s status and retrieve any return values.

When a node splits, the split-off node replicates the sequence number (in NMP-side nodes, the parent sequence number) of the original node. This ensures sequence number consistency between the host-side parent and the NMP-side children, even after node splits.
Table 6.1: Evaluation framework configuration.

<table>
<thead>
<tr>
<th><strong>Host Configuration</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host cores</strong></td>
<td>8 out-of-order* processors (ARMv7 Cortex-A15)</td>
</tr>
<tr>
<td></td>
<td>2GHz frequency, 1 thread/core</td>
</tr>
<tr>
<td></td>
<td>*Note: Some experiments were executed with</td>
</tr>
<tr>
<td></td>
<td>in-order single-cycle processors for faster simulation.</td>
</tr>
<tr>
<td></td>
<td>These are explicitly stated with the analysis.</td>
</tr>
<tr>
<td><strong>L1 cache</strong></td>
<td>32kB icache, 64kB dcache, private</td>
</tr>
<tr>
<td></td>
<td>2-way set-associative LRU</td>
</tr>
<tr>
<td></td>
<td>2-cycle latency, 128B/block</td>
</tr>
<tr>
<td><strong>L2 cache</strong></td>
<td>1MB, shared, 8-way set associative LRU</td>
</tr>
<tr>
<td></td>
<td>20-cycle latency, 128B/block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Memory Configuration</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 HMC w/ 16 memory vaults total (8 main memory vaults, 8 NMP vaults)</td>
<td></td>
</tr>
<tr>
<td>128MB/vault (total 2GB), 8 DRAM banks/vault</td>
<td></td>
</tr>
<tr>
<td>(t_{RC}): 13.75ns, (t_{RCD}): 13.75ns, (t_{CL}): 13.75ns, (t_{BURST}): 3.2ns</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NMP Core Configuration</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NMP cores</strong></td>
<td>1 in-order single-cycle processor/vault</td>
</tr>
<tr>
<td>(gem5 TimingSimpleCPU), 2GHz frequency</td>
<td></td>
</tr>
<tr>
<td>128B buffer in memory controller</td>
<td></td>
</tr>
<tr>
<td><strong>scratchpad memory</strong></td>
<td>40kB/NMP core, 8kB reserved for host memory-map, stores instructions and program stack.</td>
</tr>
</tbody>
</table>

### 6.2 Evaluation Methodology

As described in Chapter 4, because NMP-capable memory devices are not readily available yet, we rely on the Brown-SMCSim\[22\] NMP architecture simulator to evaluate our proposed hybrid data structures.

To run simulations in a reasonable amount of time, we configure a relatively small simulated system. Table 6.1 outlines the simulator configurations for this work. When the host-side processors are configured as out-of-order cores, each simulation takes approximately 24 hours to complete. For extensive sensitivity analysis (Chapter 6.4), we use even simpler host-side cores (in-order single-cycle cores – i.e., gem5’s TimingSimpleCPU module) so that the simulation execution time is further reduced to approximately 10 hours. Although the simulated system is relatively small and simple, we nevertheless show that our hybrid data structures are effective in providing high performance and reducing overall energy consumption. We also provide scalability analysis (Chapter 6.5) to project that the improvements will carry through to larger high-performance systems.

We assess data structure performance in terms of operation throughput, which is the number of data structure operations completed across all available threads in a given period of time. For our baseline analysis (Chapter 6.3), we used a pre-defined core workload from the Yahoo! Cloud Serving Benchmark (YCSB)\[28\] framework, which provides (i) keys to populate the data structure with and (ii) operations to perform on the data structure, based on distributions that appear in realistic cloud OLTP applications. The core workload we used is YCSB-C, a read-only workload with a zipfian distribution in accessed keys. We also provide a sensitivity analysis (Chapter 6.4) to show how the hybrid data structures scale with concurrent modifications. For the sensitivity analysis, we generated our own workloads with varying ratios of insertions and removals and uniform distribution of accessed keys.

We also report the system energy consumption in the baseline evaluation. The energy that we report includes:
1. energy consumed by reads and writes on the on-chip L1 and L2 caches;
2. energy consumed by data movement on the off-chip interconnect;
3. read, write, activate, and precharge energy of DRAM,
4. dynamic and static energy consumed by the scratchpad memory of NMP cores, and
5. dynamic and static energy consumed by the NMP cores.

We focus mostly on the dynamic energy consumed by the memory system, excluding the static energy of memory system components and refresh energy of DRAM, which simply increase proportionally with execution time. However, we include both the static and dynamic energy of NMP cores and associated scratchpad memories, for these are the extra hardware components that provide the basis for improved performance.

We use CACTI 7.0 [10] and assume 22nm process technology to model the energy consumption of caches, and scratchpad memory. NMP core energy is modeled using McPAT [75], also assuming 22nm technology. For the off-chip interconnect, we follow prior works [17, 39] that assume 3pJ/bit energy consumption in the HMC interconnect. For the DRAM itself, we use energy reported by the DRAMPower tool [20] integrated in gem5.

6.3 Baseline Evaluation

Skiplist

To evaluate the hybrid skiplist, we compare the operation throughputs of hybrid skiplists with blocking and non-blocking NMP calls (hybrid-blocking and hybrid-nonblocking, respectively) against the NMP-based skiplist of prior work (Chapter 5.1.2) and the lock-free skiplist as a non-NMP reference. In particular, hybrid-nonblocking\textsuperscript{4} denotes that we allowed each host thread to have up to 4 NMP calls in-flight at a time.

We initialized each skiplist with $2^{22}$ key-value pairs (generated by the YCSB framework according to core workload characteristics), which results in a skiplist of size 0.5GB with 22 total levels. We configured the hybrid skiplists to hold the top 13 levels in the host-managed portion, which makes its size roughly equivalent to the last-level cache (LLC) size.

Because there was little variation in performance and scalability across the different YCSB workloads, we base our analysis on results gathered with YCSB-C. Figure 6.4a shows the operation throughput of various skiplist implementations with YCSB-C. We plot the throughput across varying numbers of host threads to show scalability with increasing numbers of threads. Similar to results from prior work (Chapter 5.3.3), although the skiplist is significantly larger than the LLC (512\times), lock-free still shows higher operation throughput than NMP-based. However, our hybrid-blocking skiplist increases the throughput by 99\% over NMP-based and by 46\% over lock-free at 8 concurrent threads.

The performance improvement with hybrid skiplists comes from significantly reducing the number of DRAM reads. As shown in Figure 6.4b, hybrid skiplists incur only two thirds the amount of
Figure 6.4: Baseline skiplist performance evaluation with YCSB-C.

Figure 6.5: Skiplist energy consumption breakdown for baseline evaluations with YCSB-C.

DRAM reads incurred by the lock-free skiplist. More specifically, DRAM reads in non-NMP lock-free skiplists, caused by LLC misses, originate not only from CPU’s data reads but also from page table walker’s page table reads. The hybrid skiplist’s better cache utilization allows the LLC to retain page table entries to serve most TLB misses, and this also reduces the number of memory reads needed for each skiplist operation.

The performance improvement with hybrid skiplists becomes even more substantial with non-blocking NMP calls, which hide NMP operation offload costs and reduce the idle time of both host and NMP cores. At 8 concurrent threads, hybrid-nonblocking4 shows $2.46 \times$ the throughput of lock-free.

For the energy analysis, we focus on the 8-thread execution and compare the energy consumed while completing the same amount of work with the lock-free and hybrid-nonblocking4 skiplists. As described in Section 6.2, the energy analysis focuses heavily on dynamic energy consumed by
the memory system, although both the dynamic and static energy are considered for hardware components added on for NMP capabilities. We omit hybrid-blocking skiplist’s energy because hybrid-blocking and hybrid-nonblocking exhibit similar data access patterns and therefore show similar dynamic energy consumption patterns.

Figure 6.5 shows the energy consumed with each skiplist running the YCSB-C workload, under the same baseline configurations as the performance evaluation. It shows that hybrid-nonblocking consumes only half the energy compared to lock-free, even with the extra energy consumed by NMP cores and associated scratchpad memories. Most of the improvement comes from the reduction in interconnect energy consumption. The interconnect energy makes up nearly half (48%) of the total energy consumption in lock-free; by reducing the data movement between on-chip cache and off-chip memory, hybrid-nonblocking reduces the interconnect energy consumption by 97%. The reduction in DRAM energy, which makes up the other half (47%) of the lock-free’s energy consumption, is also significant, and this is proportional to the reduction in number of memory read operations. Hybrid-nonblocking reduces the DRAM energy consumption to 68% of the amount in lock-free.

B+ Tree

We compare the performance of our hybrid B+ trees with blocking and non-blocking NMP calls (hybrid-blocking and hybrid-nonblocking, respectively) against a host-only B+ tree. The host-only B+ tree is implemented with sequence locks for concurrency, just like the hybrid B+ tree, but with all levels in the host. We use this implementation as the non-NMP baseline, as it yields significantly higher throughput than more conservative concurrent B+ tree implementations.

We configured the B+ tree so that each node is 128 bytes. This is a typical B+ tree node size in in-memory OLTP systems [97], where B+ trees are popularly used for performance. In our implementation, the 128-byte node size allowed each leaf node to hold up to 14 key-value pairs and each non-leaf node to hold up to 15 children. We populated each B+ tree with approximately 30M key-value pairs and inserted the items in sorted order. This generated a balanced initial B+ tree of 9 total levels and approximately 0.57GB total size. The top 6 levels (summing up to 1.14MB in size) were placed in the host-managed portion for the hybrid implementations.

We first evaluate the B+ trees with the read-only YCSB-C, which provides the optimistic best case. Because YCSB workloads A, B, and C have the same key distributions and are guaranteed not to incur structural changes to the tree, hybrid implementations yield similar throughput across the three workloads. However, cache invalidations generated by update operations lower the throughput for host-only in workloads A and B.

Figure 6.6a shows that the hybrid-blocking B+ tree yields 18% higher throughput than host-only. While 18% increase is not insignificant, the improvement is relatively small compared to the hybrid skiplists, where hybrid-blocking yielded nearly 50% higher throughput than lock-free (Figure 6.4a).

The reason for the relatively small increase in operation throughput is in the small number of
Figure 6.6: Baseline skiplist performance evaluation with YCSB-C.

Table 6.2: NMP operation offload costs in Brown-SMCSim.

<table>
<thead>
<tr>
<th>Event</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation valid flag written from host reaching NMP core</td>
<td>65-80ns</td>
</tr>
<tr>
<td>Operation completion flag written from NMP core reaching host</td>
<td>88-96ns</td>
</tr>
<tr>
<td>Last-level cache miss</td>
<td>100-120ns</td>
</tr>
</tbody>
</table>

memory reads needed for each operation, relative to the cost of offloading operation requests to NMP cores. Figure 6.6b shows the average number of DRAM reads per operation. The host-only B+ tree makes approximately 9 DRAM reads per operation, but the hybrid B+ trees make approximately 3.5 DRAM reads, which includes both the 3 node reads made by the NMP core and the occasional DRAM reads incurred by LLC misses on the host-side. While the reduction in the number of memory reads is nearly $3 \times$, the absolute number of reads is still small, particularly compared to the skiplists, where lock-free and hybrid make 36 and 24 memory reads per operation, respectively.

On a related note, the delays in offloading operation requests to NMP cores are shown in Table 6.2. These delays were measured across multiple iterations of a single operation offload request in otherwise same settings as the baseline B+ tree evaluation (i.e., same initial B+ tree size, same number of host-side levels, and same architectural configurations). Note that the delays just for communicating operation information to and from the NMP core sum up to be comparable to 1-2 LLC cache miss delays. When the data structure operation incurs only a few memory reads, this overhead can dominate the performance of hybrid-blocking data structure implementations. However, nonblocking NMP calls effectively hide this overhead and significantly improve the hybrid B+ tree’s
At 8 concurrent threads, hybrid-nonblocking4 yields $2.11 \times$ the throughput of host-only.

We evaluate the energy consumption of B+ tree implementations in the same way that we looked into skiplist energy consumptions. Figure 6.7 shows the breakdown of the energy consumed in the baseline YCSB-C evaluation with host-only and hybrid-nonblocking4. Because of the small number of memory reads per operation, the energy consumed by NMP cores and scratchpad memories make up a relatively large portion of the total energy consumed in the hybrid-nonblocking4 B+ tree (42%, as opposed to 26% in the hybrid-nonblocking4 skiplist). Nonetheless, hybrid-nonblocking4 consumes only 48% of the energy consumed with host-only. Here, we see that both the interconnect and DRAM energy contribute significantly to the energy reduction with hybrid-nonblocking4. In host-only, the interconnect and DRAM energy each make up 47% and 46% of the total energy consumption. However, hybrid-nonblocking4 reduces the energy consumption of each component to 8% and 39%, respectively, of the amounts consumed in host-only.

6.4 Sensitivity Analysis

For the sensitivity analysis, we generated our own workloads with varying ratios of insertions and removals and uniform distribution of accessed keys. In Figures 6.8 and 6.9, X-Y-Z of the x-axis denotes the ratio of read-insert-remove operations in the workload. The initial configurations of the skiplists and the B+ trees remain the same as our baseline evaluation experiments.

In each experiment, we fixed the number of concurrent host threads to 8 (the maximum number of host threads available in the simulated system). Also, for faster simulation, the experiments were executed with TimingSimpleCPUs (in-order, single-cycle CPUs) as host cores. Note that the general trends of the results are still the same as using out-of-order host cores, for memory access is the main performance-limiting factor in these data structures, not CPU cycles.

SkipList

Figure 6.8 shows the operation throughput of the various skiplist implementations across the different workloads, normalized against the throughput of the lock-free skiplist with the 100-0-0 workload.
Figure 6.8: Skiplist sensitivity evaluation: normalized operation throughput (lock-free 100-0-0 throughput as baseline).

Although increasing concurrent modifications reduces the throughput across all skiplist implementations, it has relatively smaller impact on the hybrid skiplists. With the 50-25-25 workload, lock-free, hybrid-blocking, and hybrid-nonblocking4’s throughputs are 80%, 90% and 93% of their throughputs with the 100-0-0 workload, respectively. In other words, the hybrid skiplists have relative advantage with more concurrent modifications. At 50-25-25, hybrid-blocking and hybrid-nonblocking4 each have 1.61× and 3.12× the throughput of lock-free.

B+ Tree

The workloads for the B+ tree sensitivity analysis were generated with the following additional characteristics. While the keys accessed for read and remove operations were uniformly distributed across the entire key space, the insert keys were chosen so that insertions in the hybrid B+ trees would happen at the last leaf node (in terms of incrementing keys) of each NMP partition. This was in order to forcefully incur maximum possible node splits, while still evenly distributing the insertions across the NMP partitions.

To verify how the node splits impact performance, we also used a 50-25-25 fully uniform workload, in which even the insertions were uniformly distributed across all leaf nodes. This workload did not incur any node splits, despite the high ratio of insert operations.

Figure 6.9 shows the operation throughput of the various B+ tree implementations, normalized against the throughput of the host-only B+ tree with the 100-0-0 workload. First, we note that the throughput of hybrid-blocking decreases slightly with increasing amounts of modifications and node splits. With the 50-25-25 workload, hybrid-blocking yields 10% less throughput than the same implementation with the 100-0-0 workload; with the 50-25-25 fully-uniform workload that does not incur node splits, the throughput is 7.5% less. Even so, hybrid-blocking shows performance comparable to host-only across all workloads. With the 50-25-25 workload, where hybrid-blocking is least advantageous, hybrid-blocking still yields 93.5% of the throughput of host-only.

On the other hand, the performance of host-only slightly increases, by up to 2.7% with more modifications. This slight improvement in performance is due to better cache locality. Because the insertions in these workloads are targeted at specific leaf nodes to incur maximum node splits, nodes

---

3If operations are heavily targeted to a single NMP partition, operations will serialize in the specific partition and become the performance bottleneck. This is a limitation of the hybrid data structures that could be addressed in follow-up works.
on the insertion target paths are likely to be accessed from the on-chip cache. Figure 6.10 shows that this is indeed true: with the host-only B+ tree, the number of memory reads per operation decreases with increasing insertion ratios. The 50-25-25 fully-uniform workload eliminates the cache locality benefits, and in this case, host-only’s operation throughput decreases by 6%, compared to the read-only workload. In fact, this workload makes hybrid-blocking to have relative advantage (6% higher throughput) over host-only.

Lastly, we note that regardless of the workload, hybrid-nonblocking4 yields approximately 50% higher throughput than host-only. More specifically, the improvements in throughput range from 46%, with the 50-25-25 workload where hybrid is least advantageous, to 60%, with the 50-25-25 fully-uniform workload where hybrid is most advantageous.

6.5 Scalability Analysis

The benefits of our proposed hybrid data structures are scalable to larger data structures used in larger systems, based on the fact that hybrid data structures are designed to alleviate the performance bottlenecks of existing non-NMP data structures. More specifically, we show – through our own performance profiling and through results from related work – that non-NMP data structures suffer from frequent cache misses and long memory latencies even on systems with larger cache and even in the context of real applications.

For our own performance profiling, we ran the YCSB-C workload against a baseline lock-free skiplist and host-only B+ tree on a conventional desktop machine (configurations described in Table 6.3). For each execution, the data structure was initialized to ≈1GB in size, and 8 concurrent threads were used to run operations on the data structure. To analyze performance bottlenecks, we used the Intel VTune performance profiler.
Table 6.3: Desktop machine configuration.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel(R) Xeon(R) CPU E5-1630 v4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 cores, 2 hardware threads per core</td>
</tr>
<tr>
<td></td>
<td>3.70 GHz clock speed</td>
</tr>
<tr>
<td>Cache</td>
<td>L1I: 32KB per core, 8-way set associative</td>
</tr>
<tr>
<td></td>
<td>L1D: 32KB per core, 8-way set associative</td>
</tr>
<tr>
<td></td>
<td>L2: 256KB per core, 8-way set associative</td>
</tr>
<tr>
<td></td>
<td>L3: 10MB shared, 20-way set associative</td>
</tr>
<tr>
<td>Memory</td>
<td>32GB</td>
</tr>
</tbody>
</table>

Figure 6.11: CPU pipeline slot utilization with conventional data structure implementations.

VTune’s microarchitecture exploration analysis characterizes how the CPUs’ pipelines are utilized in program execution. Pipeline slots are categorized into:

- **retiring** (slots containing useful work; an ideal execution will have 100% retiring slots),
- **bad speculation** (slots containing work that get discarded due to bad speculation),
- **front-end bound** (slots that fail to be filled due to front-end issues, such as instruction cache misses),
- **memory bound** (slots that are stalled waiting for load/store to complete), and
- **core bound** (slots stalled because of contention on core’s compute resources).

For memory bound pipeline slots, VTune also provides submetrics on the source of stalled CPU cycles. It presents the fraction of CPU cycles spent waiting on the L1, L2, L3 cache, DRAM, or even store-bound dependencies. Because these submetrics are gathered by hardware event-based sampling, the numbers may not be as accurate as the pipeline slot characterization. Nonetheless, the general trends are valid and meaningful in identifying the major performance bottlenecks.

Figure 6.11 shows the pipeline slot utilization of the YCSB-C workload execution with the skiplist (6.11a) and the B+ tree (6.11b). With either data structure, more than 50% of the pipeline slots

Figure 6.12: Breakdown of memory bound clockticks with conventional data structure implementations.
are memory bound, even though the data structure’s size is relatively small (1GB, ≈100× the LLC size). Furthermore, Figure 6.12 shows the approximate breakdown of the memory-bound clock cycles. According to these results, approximately 80% of the memory-bound clock ticks are spent waiting on DRAM; these results confirm that conventional, non-NMP data structures are bottlenecked by DRAM access latencies, caused by last-level cache misses. Our hybrid data structures address this very issue. They are designed to improve performance by significantly reducing DRAM accesses, which is achieved by retaining frequently-accessed higher-level data structure nodes in the last-level cache.

On a related note, Sirin et al. [105, 106] extensively characterized the microarchitectural behavior of in-memory online transaction processing (OLTP) systems, which are enterprise-level applications in which high-performance hierarchical data structures are often at the core of the implementation. OLTP workloads are typified by a high volume of short-lived operations on few data items, and OLTP systems rely heavily on data structures that enable quick data lookups by index. However, Sirin et al. showed that memory latency arising from such index lookups is a significant performance bottleneck in these systems.

Their results on DBMS N4 and Silo 112 systems are particularly relevant to our work. DBMS N uses the red-black tree [11], a variant of a self-balancing binary search tree, and Silo uses Masstree [80], a custom-designed B+ tree-like trie, as their index data structures, respectively. Although the data structures are not exactly the same as the data structures that we focus on in this work, they share the hierarchical and pointer-chasing characteristics. Moreover, while other OLTP systems that they studied have legacy code ported over from prior disk-based systems, these two systems have been specifically designed ground-up for in-memory databases; thus, analysis on these systems most accurately represent the inefficiencies of in-memory OLTP.

Figure 6.13 from Sirin et al.’s recent paper [106] shows the CPU cycle breakdown of the index lookup phase of DBMS N and Silo for a 100GB database4. The Figure shows that in either OLTP

---

4The source is not identified in the Sirin et al.’s paper [106].

5Figure 6.13 also shows the CPU cycle breakdown of the transaction setup phase in DBMS N, for DBMS N spends 33% of its execution time in transaction setup. Silo does not require a separate transaction setup phase. We look into only the index lookup phase, for this is where the hierarchical data structures are involved.
system, no less than 70% of the index lookup time is stalled due to data cache misses. Furthermore, the paper also mentions that DBMS N and Silo each spend 44% and 75% of their total execution time, respectively, in index lookup. This again shows that our hybrid data structures that significantly reduce cache miss latencies can improve the performance of even large-scale in-memory OLTP systems.
Chapter 7

Impact of NMP on Memory-Hard Cryptographic Functions

This last part of my dissertation [24] presents preliminary investigations into a tangential research problem. Time-consuming and power-hungry memory-hard cryptographic functions exploit non-negligible access costs of DRAM-based main memory to hinder brute-force security attacks – how then, would these functions be affected with the advent of NMP architectures and other compute-capable memories? We first provide a brief background of memory-hard cryptographic functions, and then we focus on a single widely-used memory-hard algorithm – scrypt [92, 93] – to explore how NMP may impact its security.

7.1 Memory-Hard Cryptographic Functions

As mentioned earlier, time-consuming and power-hungry memory-hard cryptographic functions serve the purpose of hindering brute-force security attacks. A notable example of a brute-force security attack would be password cracking, in which a malicious user brute-force guesses password phrases until it finds the right password. In order to prevent such attacks, systems employ password hashing via password-based key derivation functions [62]. These functions are designed to be computationally expensive enough such that the cost of computing the hash would be negligible for an honest user, who would compute it only once, but would be significant and therefore prohibitive for a brute-force attacker, who would need to compute it repeatedly for many password candidates.

To this end, Colin Percival [93] defined the memory-hard algorithm: an algorithm that requires amount of memory approximately proportional to the number of computations to be performed for the function output. If sufficiently large amount of memory is required, not only would the compute time and power be bound by memory access, but the algorithm would also be resistant to brute-force attacks using customized hardware. Fast memory such as SRAM is expensive and takes up large chip area; therefore requiring large amounts of memory for a single hash computation limits the
amount of customized hardware that can be built in order to execute large-scale parallel attacks.

However, the security of memory-hard functions depends entirely on the nontrivial costs of memory access. Yet, various compute-capable memory technologies, including near-memory processing (introduced in Chapter 2) and processing-using-memory, that supplement memory with simple computation units have recently emerged as promising ways to address the performance and energy issues of memory access. There has been extensive research in improving application performance and reducing energy consumption using compute-capable memory, but to the best of our knowledge, no one has looked into how compute-capable memory may impact the security of memory-hard functions.

### 7.2 scrypt Overview

scrypt in particular is a sequential memory-hard password-based key derivation function, meaning that the fastest sequential algorithm to solve the function is memory-hard, and it is impossible for a parallel algorithm to asymptotically achieve a significantly lower cost. The algorithm was first proposed in 2009 and was published as RFC 7914 in 2016.

**Algorithm 1** scrypt algorithm

```plaintext
1: function Scrypt(P, S, p, N, r, dkLen)
2:   (B₀||B₁||...||Bₚ₋₁) ← PBKDF2_{SHA256}(P, S, 1, 128rp)
3:   for i = 0 to p - 1 do
4:     Bᵢ ← SMixᵣ(Bᵢ, N)
5:   return PBKDF2_{SHA256}(P, B₀||B₁||...||Bₚ₋₁, 1, dkLen)

6: function SMixᵣ(B, N)
7:   X ← B
8:   for i = 0 to N - 1 do
9:     Vᵢ ← X
10:    X ← BlockMix_{Salsa20/8,r}(X)
11:   for i = 0 to N - 1 do
12:     j ← Integerify(X) mod N
13:    X ← BlockMix_{Salsa20/8,r}(X ⊕ Vᵢ)
14: return X

15: function BlockMix_{Salsa20/8,r}(B₀||B₁||...||B₂ʳ₋₁)
16:   ▷ each Bᵢ must be 64-bytes (enforced by Salsa20/8 definition)
17:   for i = 0 to 2ʳ - 1 do
18:     X ← Salsa20/8(X ⊕ Bᵢ)
19:     Yᵢ ← X
20: return Y₀||Y₁||...||Y₂ʳ₋₂||Y₁||Y₂||...||Y₂ʳ₋₁
```

Algorithm 1 shows the scrypt algorithm as described in the original proposal and the RFC. Lines 1-4 provide the high-level flow of scrypt. Inputs *P* and *S* are password and salt phrases, respectively, and *dkLen* is the desired key length. The password and salt phrases are first passed

---

1Salt is random bits of data that is added onto a password to strengthen its security.
through \( PBKDF2_{SHA256} \) to generate a 128\( r \)-byte string. The generated string is divided into \( p \) equal-length blocks, and the SMix function is called on each of them. The results from the SMix function are concatenated back together to be used as the salt in a final \( PBKDF2_{SHA256} \) call, which takes the original password and the new salt to generate a final \( dkLen \)-byte output key.

\( p, N, \) and \( r \) are scrypt-specific parameters. \( p \) determines the number of times SMix is called in scrypt (lines 3-4). It is referred to as the parallelization parameter, for the \( p \) SMix calls are independent of one another and can be computed in parallel. \( N \) is a cost parameter passed to the SMix function; it controls the CPU and memory usage of scrypt by requiring the SMix function to compute, store, and pseudorandomly access \( N \) different BlockMix hashes (lines 5-12). \( r \) is the block size parameter that determines the size of a block that the BlockMix function operates on (lines 13-14).

The SMix function is central to the scrypt algorithm and makes up the memory-hard component of scrypt. The scrypt RFC recommends the block size parameter to be \( r = 8 \). With this parameter, the initial input block to SMix is only 1kB in size and can easily fit in cache. However, the SMix function expands this 1kB block into an array of \( N \) blocks, and the blocks are iteratively accessed in a pseudorandom order, based on the contents of the previously-accessed block. Assuming a sufficiently large \( N \), the SMix function is bound by memory access and makes up the non-trivial cost of running scrypt.

### 7.3 NMP-based scrypt Acceleration

We assume the same NMP architecture as described in Chapter but for the preliminary investigations, we utilize only a single NMP core-vault pair. We also assume that DMA functionality is available between the NMP core’s scratchpad memory and the coupled NMP vault.

As described in Section SMix makes up the memory-hard component of scrypt; thus, we offload it to the NMP core. Since \( PBKDF2_{SHA256} \) computations are not memory-hard, they are run on the host processor.

An SMix call runs entirely on the NMP core-vault pair. The host processor communicates the 128\( r \)-byte input block \( B \) and parameters \( r \) and \( N \) for SMix through the memory-mapped portion of the NMP core’s scratchpad memory. The output of SMix is also communicated back to the host via the memory-mapped region. The 128\( rN \)-byte array \( V \) generated in SMix (lines 7-9 of Algorithm 1) is stored in the NMP vault. However, the pseudorandomly chosen 128-byte block \( V_j \) (lines 11-12) is always read into the scratchpad memory prior to the bitwise-xor operation in line 12. Figure 7.1 describes the host-NMP interaction and the data placement for the SMix function in the NMP-aware scrypt implementation.

Reading the random blocks into scratchpad memory is necessary in order to reduce redundant DRAM activity that causes delays and power consumption that cannot be reduced by NMP, as

---

1 \( PBKDF2 \) iteratively applies a designated pseudorandom function on the password and salt phrases a specified number of times to generate a cryptographic key. In scrypt, SHA256 is used as the pseudorandom function and is iterated only once. SHA256 is easy to compute and is not memory-hard.
Figure 7.1: The host-NMP interaction and data placement for the $\text{SMix}$ function in the NMP-aware $\text{scrypt}$ implementation.

was identified in our prior work (Chapter 5). Because the NMP core is simple and does not have any sophisticated functionality, the bitwise-xor is expected to be executed as a sequence of simple xor instructions that operate on word-length data. Since the NMP core also does not have cache, every one of these xor instructions would incur DRAM operations to access the small portion of the block being xor-ed. Reading a word-length portion of interest from $V_j$ in memory goes through the following process: the DRAM row containing the portion is activated, the corresponding columns are selected, and then the bits are transferred to the NMP core. Each of these steps with non-negligible delays would all be repeated for every word in $V_j$, even though the DRAM row contains several contiguous words of $V_j$. Therefore, the entire block $V_j$ must be read into scratchpad memory using DMA in order to eliminate redundant DRAM row activations.

7.4 Evaluation & Discussion

Our evaluations were made on Brown-SMCSim [22] (detailed in Section 4.2.1). Table 7.1 summarizes the details of the evaluation framework.

Table 7.1: Evaluation framework details. Only one host processor and one NMP core-vault pair were used for our investigations.

<table>
<thead>
<tr>
<th>Host Configuration</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>8 in-order processors (gem5 TimingSimpleCPU)</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32kB icache, 64kB dcache, private, 2-way set-associative 2-cycle access latency, 256B/block</td>
</tr>
<tr>
<td>L2 cache</td>
<td>2MB, shared, 8-way set associative 20-cycle access latency, 256B/block</td>
</tr>
<tr>
<td>memory</td>
<td>2GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NMP configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMP core</td>
</tr>
<tr>
<td>scratchpad memory</td>
</tr>
<tr>
<td>NMP vault</td>
</tr>
</tbody>
</table>

We referred to code in the $\text{scrypt}$ git repository [108] to implement the $\text{scrypt}$ algorithm on Brown-SMCSim. Our host-based and NMP-based $\text{scrypt}$ implementations on Brown-SMCSim are available as open-source at https://github.com/jiwon-choe/Brown-SMCSim/tree/scrypt.
We compare the total execution time of *scrypt* with the *SMix* function executed on the host processor and on the NMP core. We varied the *scrypt* parameters for these measurements – Figure 7.2(a) shows the execution times with varying values of \( N \), and Figure 7.2(b) shows the execution times with varying values of \( r \). For all experiments, \( p \) was set to 1, and the desired key length was set to 64 bytes. For password and salt phrases, we used “pleaseletmein” and “SodiumChloride”, respectively, which were used to generate some of the test vectors provided in the *scrypt* RFC [92].

From the evaluation, we see that offloading the *SMix* function to NMP yields a consistent 1.5× speedup in *scrypt* execution time, regardless of the \( N \) and \( r \) values. Varying the \( p \) value will have little impact on the speedup, for an increased \( p \) would only require more NMP core-vault pairs to run in parallel. Moreover, although a consistent 1.5× speedup may not be an immediate threat to the security of *scrypt*, note that the speedup is achieved using an unsophisticated NMP hardware baseline.

In fact, parts of the *scrypt* algorithm have the potential to be further accelerated with compute-capable memory. For example, the *Salsa20/8* stream cipher [12] used in *BlockMix* (line 17 of Algorithm 1) is simply bitwise add-rotate-xor operations repeated over several rounds on a 64-byte block; furthermore, the *BlockMix* function output is just a reordering of the 64-byte output blocks from *Salsa20/8*. These functions have great potential to be accelerated with specialized near-memory accelerators or even with processing-using-memory (PUM). Computing bitwise operations in memory has been frequently explored in PUM research [2, 25, 37, 76, 103], although extending the prior PUM work to accelerate *scrypt* computation is still an open problem.
Chapter 8

Future Work & Conclusion

The core research problem of this dissertation was adapting concurrent data structures in order to fully exploit the potential of near-memory processing architectures. The solutions proposed in the dissertation consisted of (1) hardware enhancements for NMP architectures and (2) new hybrid software algorithms for the data structures. Empirical evaluations using cycle-accurate, full-system architecture simulations showed that the proposed solutions can provide significant performance gains and energy savings, compared to concurrent data structures on conventional architecture or naïve NMP-based concurrent data structure implementations.

One area of future work would be extending hybrid data structure algorithms of Chapter 6 to work with more sophisticated NMP architectures, such as ones where host and NMP cores share the memory space. Synchronization and coherence between the host and NMP cores may become more challenging in such architectures, but if there is a viable solution, the hybrid data structures may also be complemented with non-NMP accelerators that support generic data structures (e.g., QEI [122], Widx [67]).

There are also limitations to be addressed with the hybrid data structures. The most pressing issue would be improving the hybrid data structures’ performance with workloads that have high skew. Under such workloads, data structures for conventional architectures are likely to retain all relevant nodes in the on-chip cache; this will result in better performance than hybrid implementations that force lower levels to remain in memory. For hybrid data structures to yield comparable performance in such scenarios, they could be enhanced with self-adjusting algorithms (e.g., biased skiplists [9], splay-lists [5], CBTREE [1]) to dynamically push highly popular nodes to the host-managed region.

Moreover, hybrid data structure designs can be enhanced to exploit the high internal memory bandwidth offered by NMP-enabling technologies. This may require a radically new data structure design – not unlike the masstree data structure [80] – especially for the NMP-managed portion of the hybrid data structure.

Nonetheless, our work has laid the foundations for the discussed future work, and we look forward to seeing follow-up works that further enhance the effectiveness of NMP-hybrid data structures.
On another note, there are also more research areas to be explored with regards to the tangentially related dissertation work on NMP-aware memory-hard cryptographic functions. Our preliminary investigation looked briefly into the scrypt algorithm, but as mentioned in Section 7.4, scrypt has potential to be exploited further using various compute-capable memory technologies. Furthermore, scrypt is only one of many memory-hard cryptographic hash functions – Argon2 [14], Catena [35], Lyra2 [6], and yescrypt [94] are all memory-hard password hashing algorithms that received recognition in the Password Hashing Competition\footnote{https://password-hashing.net/}. In particular, Argon2 was the winner of this competition, so its security against compute-capable memory would be interesting to look into.

More recently, memory-hard algorithms are being explored not only as password hashing algorithms, but also as proof-of-work (PoW) puzzles for blockchain mining. Ethash [33] (used in Ethereum [114]), Equihash [15] (used in Zcash [53]), and Cuckoo Cycle [110] (used in Cortex [21]) are some examples of memory-hard algorithms being used as Blockchain PoW puzzles. Building accelerators for these memory-hard PoW puzzles can undermine the tamper-proof quality of blockchains, making this an even more interesting area of future work. Wu et al. [115] have proposed a memory architecture-aware accelerator design for Ethash, but further work remains in exploring the vulnerability of memory-hard functions against near-memory processing and other compute-capable memory technologies.
Bibliography


