Proceedings of

MCHPC'18: Workshop on Memory Centric High Performance Computing

Held in conjunction with

SC18: The International Conference for High Performance Computing, Networking, Storage and Analysis

Dallas, Texas, November 11-16, 2018

The Association for Computing Machinery, Inc.
2 Penn Plaza, Suite 701
New York, NY 10121-0701

ACM ISBN: 978-1-4503-6113-2

MCHPC'18: Workshop on Memory Centric High Performance Computing

Table of Contents

Message from the Workshop Organizers

Welcome to the 2018 Workshop on Memory Centric High Performance Computing!

The growing disparity between CPU speed and memory speed, known as the memory wall problem, has been one of the most critical and long-standing challenges in the computing industry. The situation is further complicated by the recent expansion of the memory hierarchy, which is becoming deeper and more diversified with the adoption of new memory technologies and architectures, including 3D-stacked memory, non-volatile random-access memory (NVRAM), memristor, hybrid software and hardware caches, etc. The MCHPC workshop aims to bring together computer and computational science researchers from industry, government labs, and academia, concerned with the challenges of efficiently using existing and emerging memory systems for high performance computing.

We would like to thank all authors who submitted papers to this workshop. Special thanks go to the program committee members for providing us with high-quality reviews under tight deadlines. For each submitted paper, we were able to collect at least three reviews. Based on the reviews, six regular papers and three short papers were selected from twelve total submissions.

We are very thankful to our Keynote speakers, Frank T. Hady from Intel and Bruce Jacob from the University of Maryland at College Park. We appreciate our panel team, including moderator Maya B. Gokhale from Lawrence Livermore National Laboratory, and panelists Mike Ignatowski from AMD, Jonathan C. Beard from Arm, Frank Hady from Intel, Bruce Jacob from University of Maryland at College Park, and Michael A. Heroux from Sandia National Laboratories. Our special thanks to ACM for publishing the proceedings of the workshop. We would also like to acknowledge the financial support of Sandia National Laboratories for publication of the workshop proceedings. It has been a pleasure to work with SC'18 Workshop Chair Guillaume Aupy and the Linklings support team on the logistics of the workshop. Last but not the least, our sincere thanks are due to the attendees, without whom this workshop would not be a success. We hope you enjoy the program!

Yonghong Yan, Ron Brightwell, Xian-He Sun, and Maya B. Gokhale
MCHPC'18 Workshop Organizers

Morning Keynote:

Converging Storage and Memory

Frank T. Hady, Intel

Abstract:
Order of magnitude advances in non-volatile memory density and performance are upon us bringing significant systems level architecture opportunities. The NAND Memory transition to 3D and the introduction of QLC have recently increased NAND SSD storage density at a very rapid pace. Products featuring one terabit per die are available from Intel® Corporation allowing dense storage, for example one PByte in 1U. This large improvement in density brings great value to systems, but also increases the performance/capacity/cost gap between DRAM and storage within the long evolving memory and storage hierarchy. Intel® 3D XPoint™ Memory, with much higher performance than NAND and greater density than DRAM has entered the platform to address this gap - first as SSDs. These Intel® Optane™ SSDs are in use within client and data center platforms as both fast storage volumes and as paged extensions to system memory delivering significant application performance improvements. With low latency and fine-grained addressability, this new memory can be accessed as Persistent Memory (PM), avoiding the 4kByte block size and multiple microsecond storage stack that accompany system storage. This Intel® Optane Data Center Persistent Memory is made possible through a series of hardware and software advances. The resulting high capacity, high performance, persistent memory creates opportunities for rethinking algorithms to deliver much higher performance applications. This presentation will explain these new memory technologies, explore their impact on the computing system at the architecture and solution level, and suggest areas of platform exploration relevant to the HPC community.

Speaker: Frank T. Hady, Intel
Frank T. Hady is an Intel Fellow and the Chief Systems Architect in Intel's Non-Volatile Memory Solutions Group (NSG). Frank leads architectural definition of products based on both Intel® 3D XPoint™ memory and NAND memory, and guides research into future advances for these products. Frank led the definition of the first Intel Optane products. Frank maintains a platform focus, partnering with others at Intel to define the deep integration of Intel® Optane™ technology into the computing system's hardware and software. Previously he was Intel's lead platform I/O architect, delivered research foundational to Intel's QuickAssist Technology, and delivered significant networking performance advances. Frank has authored or co-authored more than 30 published papers on topics related to networking, storage, and I/O innovation. He holds more than 30 U.S. patents. Frank received his bachelor's and master's degrees in electrical engineering from the University of Virginia, and his Ph.D. in electrical engineering from the University of Maryland.

Afternoon Keynote:

All Tomorrow's Memory Systems

Bruce Jacob, University of Maryland at College Park

Abstract:
Memory and communication are the primary reasons that our time-to-solution is no better than it currently is … the memory system is slow; the communication overhead is high; and yet a significant amount of research is still focused on increasing processor performance, rather than decreasing (the cost of) data movement. I will discuss recent & near-term memory-system technologies including high-bandwidth DRAMs and nonvolatile main memories, as well as the impact of tomorrow's memory technologies on tomorrow's applications and operating systems.

Speaker: Bruce Jacob, University of Maryland at College Park
Bruce Jacob is a Keystone Professor of Electrical and Computer Engineering and former Director of Computer Engineering at the University of Maryland in College Park. He received the AB degree in mathematics from Harvard University in 1988 and the MS and PhD degrees in CSE from the University of Michigan in Ann Arbor in 1995 and 1997, respectively. He holds several patents in the design of circuits for electric guitars and started a company around them. He also worked for two successful startup companies in the Boston area: Boston Technology and Priority Call Management. At Priority Call Management, he was the initial system architect and chief engineer. He is a recipient of a US National Science Foundation CAREER award for his work on DRAM, and he is the lead author of an absurdly large book on the topic of memory systems. His research interests include system architectures, memory systems, operating systems, and electric guitars.

Panel:

Research Challenges in Memory-Centric Computing

Moderator: Maya B. Gokhale, Lawrence Livermore National Laboratory

Panelist:

1. Mike Ignatowski, Sr. Fellow, Advanced Memory and Reconfigurable Computing - AMD Research
2. Jonathan C. Beard, Staff Research Engineer - Arm HPC | Future Memory/Compute Systems
3. Frank T. Hady, Intel
4. Bruce Jacob, University of Maryland at College Park
5. Michael A. Heroux, Sandia National Laboratories

Challenges of High-Capacity DRAM Stacks and Potential Directions

Amin Farmahini-Farahani, Sudhanva Gurumurthi, Gabriel Loh, Michael Ignatowski
AMD Research, Advanced Micro Devices, Inc.
{afarmahi, sudhanva.gurumurthi, gabriel.loh, mike.ignatowski}@amd.com

ABSTRACT

With rapid growth in data volumes and an increase in number of CPU/GPU cores per chip, the capacity and bandwidth of main memory can be scaled up to accommodate performance requirements of data-intensive applications. Recent 3D-stacked in-package memory devices such as high-bandwidth memory (HBM) and similar technologies can provide high amounts of memory bandwidth at low access energy. However, 3D-stacked in-package memory have limited memory capacity. In this paper, we study and present challenges of scaling the capacity of 3D-stacked memory devices by stacking more DRAM dies within a device and building taller memory stacks. We also present potential directions and mitigations to building tall HBM stacks of DRAM dies. Although taller stacks are a potentially interesting approach to increase HBM capacity, we show that more research is needed to enable high-capacity memory stacks while simultaneously scaling up their memory bandwidth. Specifically, alternative bonding and stacking technologies can be investigated as a potentially major enabler of tall HBM stacks.

CCS CONCEPTS

• Hardware~Dynamic memory

KEYWORDS

3D Stacking, Capacity, DRAM, HBM, Memory

ACM Reference format:

Amin Farmahini-Farahani, Sudhanva Gurumurthi, Gabriel Loh, and Michael Ignatowski. 2018. Challenges of High-Capacity DRAM Stacks and Potential Directions. In *Proceedings of ACM Workshop on Memory Centric High Performance Computing (MCHPC'18). ACM, New York, NY, USA, 10 pages.* https://doi.org/10.1145/3286475.3286484

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286484

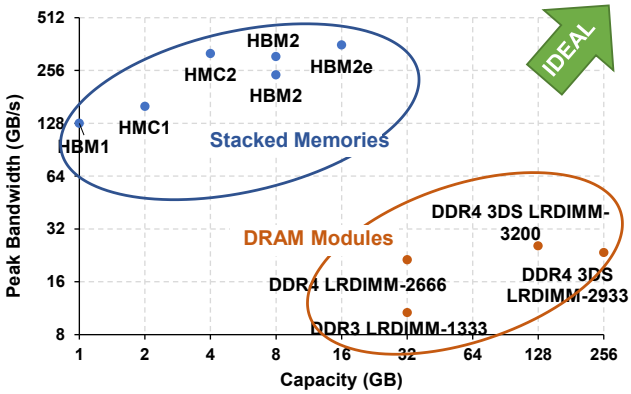


Figure 1. Bandwidth and capacity of DRAM stacks and commodity DDR DIMMs.

1. Introduction

Current CPUs/GPUs are composed of several cores and the current trend in increasing the number of throughput cores per chip is likely to continue for the foreseeable future. As data sizes grow exponentially, future GPUs and high-performance CPUs require high-bandwidth, energy-efficient access to a large memory capacity to efficiently process data-intensive applications such as scientific computing, in-memory data analytics, and artificial intelligence. Traditionally, CPUs/GPUs have relied on off-package commodity DDR and GDDR DRAM. Recently, stacked memory has been integrated into the CPU/GPU package to improve performance, bandwidth, and energy efficiency.

In-package stacked memory provides higher bandwidth and lower power consumption than off-package commodity memory for several reasons including physical proximity to the processor, wider and shorter connections to the processor, and localized accesses to a single location in a single die. However, the current generation of in-package stacked memory can suffer from low capacity. Figure 1 compares the bandwidth and capacity of commercial stacked memories such as High-Bandwidth Memory (HBM) [1] and Hybrid Memory Cube (HMC) [2] with those of high-end commodity DDR DIMMs. For instance, a single HBM2 stack currently in volume production can provide a memory bandwidth of

around 256GB/s which is almost an order of magnitude higher than the bandwidth of a high-end, state-of-the-art DDR4 DIMM. Nevertheless, memory stacks have a limited capacity. For example, a single HBM2 stack can currently provide a capacity of up to 8GB (gigabyte), containing eight DRAM dies each having a density of 8Gb (gigabit). Unlike current stacked memory devices, existing DDR DIMMs offer much higher memory capacity by populating DRAM dies in parallel on 64 or 72 bits wide DIMM (each DRAM die itself could be 3D-stacked [3]).

Higher capacity memory stacks are highly desirable as they can accommodate larger or entire application data sets in the in-package memory close to processor, accelerating application execution and decreasing the energy from data movement/migration between in-package memory and off-package memory or storage [4]. GPUs and accelerators use in-package stacked memory for graphics, high-performance computing, and artificial intelligence applications. These applications drive the need for high-capacity in-package memory as they require both high memory bandwidth and high memory capacity.

There are four approaches to improve the capacity of in-package memory. The first approach is to use higher density DRAM dies within the memory stack by integrating more DRAM cells in a single die. Higher density DRAM dies may be achieved through use of future process technologies with smaller DRAM cells. Scaling DRAM cells is projected to continue, but at a slowing pace as Moore's Law fades, exposing reliability challenges and limitations in the manufacturing process. Moreover, DRAM die area can be increased to achieve higher die capacity, possibly at the expense of reduced die yield and added cost. Larger DRAM die area will also constrain the number of HBMs that fit in the processor package. Another way to improve the density of DRAM dies is to move logic circuits (e.g., decoders) from DRAM dies to the base die of the stack and design DRAM dies that are mainly made of DRAM cells. As a result, a higher fraction of the die area is devoted to DRAM cells. These DRAM dies would potentially benefit from improved density and yield over the conventional DRAM dies that integrate logic circuits and DRAM cells into a single die. The main drawback to this design is the need for a much larger number of vertical connections to transfer signals from the base die to the DRAM dies.

The second approach to improve the capacity of in-package memory is to use high-capacity non-volatile memory (NVM) dies. Emerging NVM technologies typically provide better cell density than DRAM technology. However, NVM technologies usually face issues including lower bandwidth, larger write energy, narrow operating temperature margin, and limited write endurance. To provide a trade-off between memory capacity and bandwidth, NVM and DRAM dies can be integrated in a single stack to form a hybrid DRAM-NVM stack. As a result, this approach complicates the stack design even further and requires design from the ground up and careful thinking due to manufacturing and data management issues.

The third approach to improve in-package memory capacity is to populate a larger number of in-package memory stacks around the processor. Although this approach can improve both in-package memory capacity and bandwidth, it has cost implications. Additional memory stacks require larger interposer and package substrate which increase packaging cost. Note that the size of the interposer can be limited to the maximum reticle size. More severely, more interposer interconnects are needed between the processor and memory which further increases packaging cost and increases processor die manufacturing cost due to higher off-chip I/O counts [5].

The fourth approach is to increase the number of DRAM dies in a stack, realizing a tall stack[1]. While other approaches to improve capacity are feasible, in this paper, we this consider this approach as it is a natural evolution of the in-package stacked memory and is orthogonal to the previous three approaches. Integrating more DRAM dies within a stack, however, faces several challenges. In this paper, we outline major challenges to enable tall memory stacks and directions to address the challenges. Nonetheless, we believe more research and engineering are needed to bring taller memory stacks to the market. By presenting immediate and major challenges of high-capacity in-package memory, we hope to motivate the need for more research in several areas. We also point out some potential directions to initiate this research.

2. HBM Overview

This section provides a primer on HBM, discussing the overall design and architecture based on published JEDEC standards [1] and commercially available parts [6].

HBM is a memory standard for stacks of DRAM dies. The first and second specifications of the HBM standard were ratified by JEDEC in 2013 and 2015, respectively. HBM, as the name implies, primarily provides high amount of memory bandwidth by exploiting a large number of I/O pins

[1] By tall stacks in this paper, we mean memory stacks with more than 8 stacked DRAM dies without any notion of their physical vertical height.

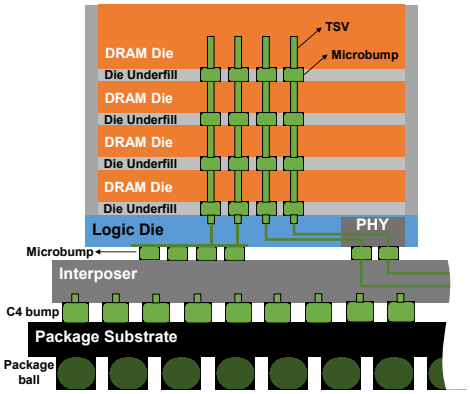


Figure 2. Stacked DRAM dies, TSVs, microbumps, and underfill in an HBM stack. The HBM stack is mounted on an interposer (figure not drawn to scale).

and therefore targets systems with high bandwidth requirements. Besides higher bandwidth, HBM also provides reasonable capacity per unit area and better energy efficiency compared to the conventional DDR DRAM external to the processor package. The access latency of HBM is comparable to that of the conventional DDR DRAM technology.

In the current HBM standard, HBM2, there are up to 8 independently-controlled memory channels. Channels in HBM2 and DDR DIMMs use a double data rate I/O interface, meaning that each pin transfers two bits of data per clock cycle. However, in contrast to latest DDR standards, each HBM channel uses a 128-bit wide I/O interface with each pin operating at a data rate of ~2Gbps (slightly more or less depending on the implementation variation). Assuming a data rate of ~2Gbps per pin, the total bandwidth of a single HBM2 channel can thus be 32GB/s and be 256GB/s for the stack. Each channel has also 8-16 banks, providing a large opportunity for memory-level parallelism. The capacity of each DRAM die is currently up to 8Gb and each stack can have up to eight DRAM dies, providing a total capacity of up to 8GB per stack. HBM2 sports several new features that are not present in conventional DDR standards. Among those is the single-bank refresh feature which refreshes only a single bank per channel at a time as opposed to refreshing all banks during refresh cycles. This allows access to other banks in a channel even if a bank is currently undergoing refresh. Another interesting feature is the pseudo channel mode which divides a channel into two individual sub-channels with separated banks. This feature provides higher bandwidth efficiency for non-streaming, irregular memory references with low spatial locality by allowing more activations in a time window.

An HBM stack is composed of a multitude of DRAM dies and a single logic die (base die). The logic die is designed to test the stack and provides the communication interface, commonly referred to as the "PHY", to the outside world. In the current HBM technology, the dies in a stack are connected by through-silicon vias (TSVs) and microbumps as shown in Figure 2. The TSVs and microbumps not only transfer data signals between dies, but also connect the dies to power and ground pins. TSVs pass through DRAM dies, providing electrical connection between the DRAM dies and logic die. In the current HBM technology, the aggregate internal bandwidth of data TSVs is the same as the external bandwidth of I/O pins. TSVs are usually copper with a diameter of 5-10µm. Microbumps join adjacent dies together and usually have a diameter of ~25µm and a pitch size of ~55µm in HBM2. The microbumps internal to the stack have the same mechanical features as bumps used in connecting the stack to a package substrate but are much smaller. Nevertheless, microbumps occupy a considerable amount of die area since many microbumps are required for data signals, power supply, testability, and mechanical stability. For instance, in the current HBM standard, all microbumps are located in the center of the stack and occupy an area of close to 20mm^2 in each DRAM die [1].

Figure 2 shows how DRAM and logic dies are conceptually stacked in the current HBM technology. All DRAM dies are usually thinned to a thickness of ~50µm (apart from the top DRAM die) [7], their functionality and performance are tested, then the good DRAM dies are stacked. Underfill material is filled in the space between adjacent dies for mechanical stability and has a thickness of ~30µm [7]. Underfill is made up of dielectric material that does not have good thermal conductivity, causing heat extraction issues in HBM. This issue becomes even more pronounced in HBMs with a higher number of stacked dies. Another related point is the physical height of the stack that may cause some packaging issues. The height increases even further in HBMs with a higher number of stacked dies if the same stacking technology is used for manufacturing future HBMs.

Figure 2 shows an HBM stack that is mounted on an interposer. The processor (not shown) is also mounted on the interposer and placed adjacent to the HBM stack in a single package. The interposer provides low-energy, high-bandwidth connectivity between the stack and processor through wide, short-distance connections.

3. Challenges of Tall Memory Stacks and Potential Directions

Stacking more DRAM dies within a memory stack (e.g., 16 stacked DRAM dies) faces key challenges. We provide a

technical overview of these challenges and some potential directions to tackle them.

3.1 TSV Speed

One of the main challenges to increasing DRAM layers within a memory stack is the speed limitation of TSVs. As the number of layers increases the maximum rate in which data can be transferred through the combined stack of TSVs degrades due to limited driver strength and capacitance which increases with the number of layers. In other words, the data transfer delay through TSVs increases with the stack height, constraining data communication bandwidth through TSVs.

The TSV delay is a function of the TSV resistance R and capacitance C (RC delay). TSVs incur non-trivial RC delay in existing stacked memories. The delay characteristics of a TSV are different from those of a metal wire because the cross-sectional area and geometrical shape of a TSV are larger than those of a wire, even those of top metal wires. The TSV resistance is a function of its material resistance (e.g., copper versus tungsten), its diameter (cross-sectional area), and its length. Figure 3 shows an abstract diagram of a TSV. The TSV resistance is directly proportional to its length, but inversely proportional to its cross-sectional area. TSVs usually have small resistance compared to short wires due to their large cross-sectional areas.

The TSV capacitance is a function of multiple parameters including TSV diameter and length, the thickness of oxide insulator liner surrounding the TSV, and the bonding technique used (Section 4). The TSV capacitance is directly proportional to its diameter and length, but inversely proportional to the oxide thickness. TSVs usually impose large capacitance due to their large size. Reducing TSV diameter and length can significantly reduce the TSV capacitance and delay, but both are influenced largely by the bonding and thinning process. Increasing the oxide thickness can reduce the RC delay as well but it can impose even more area overhead and can degrade TSV density.

Potential directions: With an increase in the number of DRAM layers, the TSV RC delay is aggravated. There are a few potential directions/workarounds for mitigating the high TSV RC delay in tall memory stacks. A potential direction is wider vertical buses within the stack to transfer more bits in parallel but at a reduced frequency of data transfer over TSVs. For example, a 256-bit wide data bus at 2GHz provides the same theoretical bandwidth as a 128-bit wide data bus at 4GHz. Wider TSV data buses, however, increase TSV area overhead. Reducing TSV diameters and pitch, if viable in the current/future stacking technology, could keep the area overhead of wider buses low. An alternative direction is to use large TSV drivers and repeaters in the path that can help reach the target TSV speed, but that imposes energy, area, and cost overheads. Finally, a research direction is to insert data buffers to buffer TSV data in intermediate layer locations and resend the data [8]. This basically breaks a long TSV data bus into shorter, but faster TSV data buses which can also help reduce the TSV delay and improve the TSV data rate. However, buffers take area and can impose non-trivial buffering delay and energy overheads.

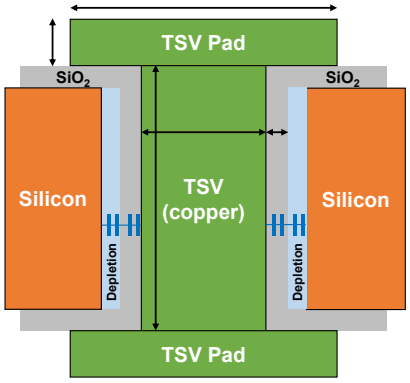


Figure 3. Abstract diagram of a TSV (not to scale).

A fundamental direction to minimizing the TSV delay is to use a different bonding and stacking technology that would potentially enable much shorter and thinner TSVs (Section 4). This type of stacking technology could greatly reduce the coupling capacitance of a TSV, thereby reducing delay. A new stacking technology that works well with DRAM requires substantial research and development.

Summary: Degradation in the TSV speed is a major challenge in tall memory stacks. The TSV speed in current HBM technologies is already at it limits, just enough to match the external bandwidth while minimizing TSV area overhead. As the number of DRAM layers increases, both TSV resistance and capacitance increase, further limiting the TSV speed. For example, going from an 8-high stack to a 16-high stack significantly increases the TSV RC delay for the top die. This means that bandwidth of stacked memories could be negatively affected by the number of DRAM layers (i.e., capacity). We expect that upcoming HBM technologies can support higher internal bandwidth using wider data TSV buses while mitigating their area overhead through marginal improvement in TSV pitch size. We, however, do not expect significant improvement in per-TSV data rates with the current stacking technology, especially for taller stacks. Future stacking technologies could potentially improve TSV speed

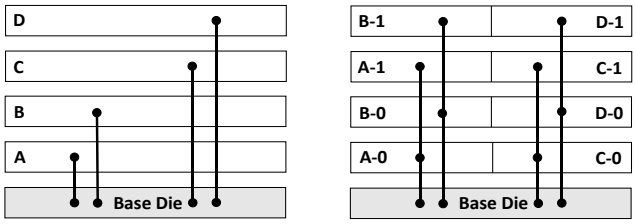


Figure 4. Example mapping of TSVs to DRAM dies in a 4-high stack: point-to-point TSV structure (left) and shared TSV structure (right). The notation of X-# represents channel X and bank group #.

by fundamental change in the TSV physical structure, but high-volume manufacturing feasibility is yet to be seen.

3.2 TSV Count

Having higher DRAM die count not only improves the capacity, but also presents the opportunity of having a higher number of channels (and thus higher bandwidth within a stack) and/or a higher number of banks per channel (better bank-level parallelism). Channels have independent address, command, and data signals. As a result, adding more channels requires incorporating more TSVs in the stack if channel width remains intact. However, adding more TSVs imposes higher area overhead.

There are two types of structures for mapping TSVs to DRAM dies: point-to-point TSV structure and shared TSV structure [9, 10, 11].

In the point-to-point TSV structure (Figure 4 left), each TSV provides a connection between a *single* DRAM die and the base die. In other words, each TSV is dedicated to a single DRAM die[2]. Some configurations of HBM uses this structure. Since in this structure each DRAM die uses its own set of TSVs, the bandwidth of a stack improves with an increase in DRAM die count provided the TSV width and frequency remain intact. In other words, capacity improvement through taller stacks goes hand in hand with bandwidth improvement. The downside is that as the number of DRAM dies within a stack increases, more TSVs are required to accommodate higher DRAM die count and higher bandwidth. Thus, this structure imposes more TSV area overhead with the increase in DRAM die count. One way to mitigate this TSV area overhead is to narrow the TSV bus in the point-to-point TSV structure (e.g., having a 64-bit wide TSV interface instead of a 128-bit interface), which negatively impacts the bandwidth delivered by each DRAM die.

In the shared TSV structure (Figure 4 right), each TSV provides a connection between *multiple* DRAM dies and the base die. For instance, in Figure 4 (right), two DRAM dies in a 4-high stack share a set of TSVs. At any given time, only one of the connecting DRAM dies can transfer data through the TSV interface. The main challenge with this shared TSV structure is data conflict over the TSV interface. No dies that share the same TSV interface can transfer data simultaneously. To avoid the problem of data conflict over the TSV interface, the memory controller must be mindful of bank and channel distribution over DRAM dies to be able to carefully orchestrate data transfers over the TSV interface without causing data conflict. Also, the switching/interleaving delay is inserted when the data transfer is switched from one die to another, causing an idle cycle in the TSV data connection and thus reducing bandwidth efficiency [12].

The shared TSV structure may provide less bandwidth than the point-to-point TSV structure as the TSV interface is shared among DRAM dies. However, judicious physical division of each channel among multiple dies and assigning each die a subset of the channel's banks would result in similar theoretical bandwidth as the point-to-point structure with the same number of TSVs. In other words, if the TSV interface is shared among banks belonging to the same channel, no bandwidth loss is caused provided no switching delay is incurred. The reason is that at any given time only a single bank within a channel transfers data over the TSV interface and no banks within a channel are permitted to transfer data simultaneously. For example, both structures in Figure 4 provide the same theoretical bandwidth assuming both structures have identical TSV widths and frequencies.

The shared TSV structure can be used to reduce TSV count at the expense of bandwidth reduction. This shared TSV structure can form logical ranks within a stack in which multiple ranks share the same TSV interface but only a single rank can transfer data at a time. 3DS DRAM [3] is an extreme example of this structure where each DRAM die is a logical rank and all DRAM dies share a single TSV data interface. Thus, the shared TSV structure presents the opportunity of improving capacity without the overhead of additional TSVs and independently of the stack internal bandwidth.

Summary: The TSV-to-DRAM die mapping structures and internal organization of the stack provide different design trade-offs in terms of capacity, bandwidth, and TSV area and cost overheads. As the number of DRAM layers within a

[2] Note that a DRAM die can be composed of more than one channel. Therefore, each channel has its own dedicated set of TSVs.

stack increase, more TSVs may be needed to scale up bandwidth. With the increase in DRAM layers, the point-to-point TSV structure provides a means to improve both capacity and bandwidth at the expense of more TSVs, while the shared TSV structure can provide a means to improve capacity and manage the TSV overhead at the expense of a worse bandwidth-to-capacity ratio. For tall memory stacks, TSV mapping structures can be investigated.

3.3 Stack Height

The physical z-height of a stack has a direct impact on packaging cost and cooling effectiveness. In addition, package thickness can be a major constraint in the mechanical design. In the latest generation of HBM technology, HBM2, the height of an 8-high stack is 700-800µm [1, 13]. Increasing the number of DRAM layers (and thus requiring underfill between DRAM layers) in a stack would further increase the height. The height of the interposer in 2.5D interposer systems or the height of the processor in 3D processor-memory systems further increases the total package height. The height of memory stacks could be a limiting factor in high-end GPUs and server-class systems as well as in mobile and embedder systems. In hand-held systems, the number of stacked dies may be shorter but the thermal and cooling constraints may be more severe. The height of future stacked memories could potentially pose packaging and thermal conductivity issue.

Potential directions and summary: In the current HBM stacking technology, we project that the height could be limiting for 16-high stacks due to limitations of DRAM die thinning and die underfill thickness. Alternative bonding and stacking techniques for DRAM could be adopted to reduce the stack height. Those techniques can present opportunities to thin DRAM dies, forgo microbumps and underfill between adjacent stacked dies, and use better thermally conductive practices (See Section 4).

3.4 Stack Thermal Conductivity

The temperature distribution within a stack depends on factors including heat sources in the stack, heat source near by the stack, package, thermal interface material, coolant flowrate, temperature, etc. But a major factor is the thermal conductivity within the stack itself. Thermal conductivity of a stack is a major challenge for cooling memory stacks and becomes more important as the number of DRAM layers and thus thermal density increases. With the increase in DRAM layers, the temperature difference between the bottom (hottest) die and top (coolest) die increases, requiring thermal conductivity improvements in the stack to enable transfer the heat from bottom to top without exceeding die temperature limits. Stacking DRAM atop a processor not only increases the temperature difference among the processor and DRAM dies but can also cause hotspots in the bottom and near-bottom dies. Even in 2.5D packages where a memory stack is beside the processor, hot spots can occur in the locations adjacent to the processor die.

The maximum die temperature specification of most DRAM devices has to be 85℃ to maintain the specified refresh rate. At higher DRAM die temperatures, higher refresh rates are required to maintain data integrity (which negatively affect available bandwidth). In addition, operating DRAM cells at temperatures beyond their specified operating temperature range leads to higher mean time between failure (MTBF) rates [14].

Potential directions: DRAM must ensure that data integrity is maintained under all allowed working conditions. To maintain data integrity in the presence of temperature disparity both within a die and across dies in a stack, adaptive refresh techniques [13] can be used to enable different refresh rates for different locations in the stack based on their temperature. This technique also presents an opportunity to save energy by lowering the refresh rate of cooler locations instead of refreshing the entire stack based on the worst-case situation.

The effectiveness of cooling a stack is directly related to thermal conductivity of the stack which depends on multiple factors including bonding technique, underfill material, die thickness, and TSV distribution. To improve cooling, researchers have proposed techniques such as dummy microbumps and thermal TSVs [15, 16], microfluidic channels [17], thermal-aware placement [18], and thermal-aware data compression [19].

Summary: Thermal conductivity can affect reliability and performance of memory stacks, especially those with more than 8 DRAM layers. As a result, improving thermal conductivity of future tall memory stacks is crucial. With existing bonding techniques, more dummy microbumps are integrated within the stack to marginally improve thermal conductivity. Alternative bonding and stacking techniques can be investigated as they can substantially improve thermal conductivity of tall memory stacks (Section 4).

3.5 Reliability

Memories can experience faults and cause reliability challenges especially as DRAM technology is becoming less stable due to extreme cell scaling. Reliability techniques are re-

quired to ensure fault tolerance within acceptable performance limits as well as high yield within acceptable area overhead. In traditional DDR DIMMs, data bits come from multiple DRAM dies. On the other hand, in HBM all data bits of a transaction are retrieved from a single row of a single bank of a single DRAM die. Data retrieval form a single location in HBM improves the energy efficiency of HBM devices over traditional DDR DIMMs. However, HBM's single-location access has a negative impact on reliability of HBM devices as faulty data bits cannot be re-constructed from other dies. Hence, some traditional reliability techniques such as ECC and chipkill, which uses a group of independent memory dies to protect against any single die failure and multi-bit errors in a die, cannot be applied directly to HBM. In addition, stacked memories introduce new elements such as TSVs that could be a new source of failure. Stacked memories are usually placed in close physical proximity to processors which could lead to thermal and mechanical stresses. These factors are a challenge to stacked memory attaining the reliability of traditional DRAM DIMMs [20].

In general, there are three types of faults in stacked memories: DRAM cell faults, peripheral logic faults, and TSV faults. While cell faults can usually cause single-bit errors, the peripheral logic and TSV faults are single points of failures that can cause multi-bit errors. For example, a single data TSV fault can cause up to four error bits when a 512-bit cache line is transferred over 128 TSVs. Address TSV faults and peripheral logic faults can cause bank, row, and column failures.

Replacing faulty stacked memory co-packaged with a processor is more costly than replacing a faulty DIMM because the entire processor-memory package must be replaced (i.e., the field replaceable unit is the processor module which costs far more than a DIMM). In addition, as the number of DRAM layers increases, the fault rate of a stack can degrade as more elements and cells are integrated in the stack. Higher fault rate of memory stacks may cause more frequent replacement of processor-memory package without memory fault-tolerant schemes.

Potential directions: In future stacked memories, more robust offline and/or online test and repair schemes could be used to detect and repair a variety of faults before causing failure and enhancing yield. Stronger error detection and correction schemes could also be used to detect and potentially correct multi-bit errors if additional ECC bits and pins can be added to future stacked memories. Finally, redundant storage in the base die of a memory stack could enable new fault-tolerant schemes.

Summary: Stacked memories face several reliability challenges especially in future DRAM stacks due to the inherent reliability issues of highly-scaled DRAM cells and an increase in the DRAM die count. Therefore, reliability techniques are required to mitigate reliability degradation for tall stacks. The area/bandwidth/energy overheads of reliability techniques can be carefully examined.

3.6 Cost and Volume

Cost and volume production of stacked memories are closely related. High volumes reduce cost, and lower costs help realize higher volumes by enabling stack memory to be adopted by larger markets. If stacked memories are adopted in consumer markets, the capital investment needed for stacked memory production and use are amortized over high volume, which can decrease cost.

Currently, stacked memory costs more than conventional memory due to lower volumes, TSV area overhead, and higher manufacturing cost (e.g., die stacking and thinning).

Potential directions: Manufacturing volumes have a first-order impact on cost, and high cost has restricted stacked memories to high-end graphics and high-performance computing markets. To increase volume and lower production cost, techniques such as reducing the number I/O pins for stacked memory devices and enabling use of organic substrates rather than interposers can be evaluated.

3.7 Power Delivery

Power delivery to in-package stacked memories, especially when 3D stacked on top of a processor, could be a design challenge [21]. Failing to deliver stable power to memory stacks may jeopardize functionality or performance targets. In addition, in-package memory stacks with high power consumption not only require more sophisticated power delivery mechanisms, but they also leave less power for the processor since they take a higher portion out of a fixed processor-memory power budget.

For sufficient and stable power delivery throughout the stack, a large number of pins and TSVs are used for power and ground in the current generation of HBM stacks. The number of pins and TSVs have a direct impact on packaging cost and DRAM die area.

In addition to delivering power to memory stacks, power should properly be distributed within the stack. Stacked memories, like other integrated circuits, must deal with IR-drop in which voltage drops over long, resistive conducting wiring traces which cause different locations in a die and stack to receive different voltages. For example, HBM2 core

Table 1. Stack internal bonding techniques and their characteristics

Technique	Requirements	Microbump/TSV pitch size	Die layer thickness	Processing thermal budget	Heat extraction capacity
Microbump (solder)	Microbump + TSV landing pad + Underfill	55µm	50µm (plus 30µm thick underfill)	~250^{0}C for a few (2-3) minutes	Poor due to underfill
Hybrid	Direct electrical connection	2.5µm for wafer-2-wafer	5µm-20µm	~400^{0}C for an hour	Very good
Direct Oxide	TSV after bonding (TSV last)	15-20µm	5-20µm	~150^{0}C for an hour	Very good

has a typical V_{dd} of 1.2V and the minimum allowed V_{dd} of 1.14V. Thus, HBM2 can only cope with an IR-drop of 0.060V. Any larger IR-drop (caused, for example, by a lossy power supply) may result in incorrect functionality (e.g., not meeting timing constraints), reduced performance (e.g., constraining data transfer rate), and/or data errors.

DRAM stacks face power delivery challenges will be exacerbated in future taller stacks. While traditional 2D circuits may suffer from horizontal IR-drop in a single die, vertical IR-drop can also be considered especially for taller stacks as some dies have longer paths to the power supply. For instance, the top DRAM die receives lower voltage than bottom dies since power is supplied through the base logic die or bottom DRAM die [22]. In addition, as the number of DRAM dies increases, the power delivered must increase to supply the additional DRAM die in the stack and thus current into the stack increases proportionally as well. The resistance of the power delivery network (PDN) increases as well since the network becomes larger. The probability of simultaneously switching noise (SSN) also increases since more banks/channels/pseudo-channels may trigger switching more wires/TSVs in unison. We also expect future generations of stacked memories to have a lower nominal supply voltage to improve power efficiency and smaller TSV sizes to improve TSV density. The former causes less tolerance to IR-drop because of smaller voltage margin and the latter causes more resistance in the PDN.

Potential directions: Designing a more effective PDN by distributing power and ground (PG) TSVs in the die could mitigate the IR-drop problem. For example, it has been shown that placing some PG TSVs on the edge of the DRAM die and some PG TSVs in the center can help reduce power supply noise in memory stacks [23]. However, distributing PG TSVs across the die imposes considerable impacts on area due to large keep-out zones of TSVs. Incorporating more PG TSVs in a centralized location could help with power delivery to some degree but it also comes at the expense of area and cost. Similarly, including and placing a sufficient number of voltage regulators and decoupling capacitors in proper locations could alleviate dynamic IR-drop caused by DRAM activity at the expense of area and cost [24], but they do not resolve static IR-drop caused by the resistance of the PDN.

Summary: Power delivery is an important design challenge in stacked memories and becomes even more important with taller stacks due to an increase in power density. The effectiveness of potential techniques for power delivery as well as their cost can be evaluated carefully.

4. Alternative Bonding and Stacking Techniques

Die bonding is the process of attaching dies to one another or to a substrate to provide electrical and physical connectivity between dies. The bonding process is required to manufacture 3D die stacks such as DRAM stacks. Table 1 describes the characteristics of the main bonding techniques for use within 3D stacks.

The conventional die bonding technique used in HBM is microbump bonding in which metal microbumps are used to bond stacked dies and establish connectivity between dies. The space between dies is filled with underfill to ensure mechanical robustness (Figure 2). The main disadvantages of microbump bonding are (1) large microbump sizes which degrade TSV density and (2) thick die and underfill which increase the stack height. The underfill also has a negative impact on the stack thermal conductivity. The height and diameter of TSVs are usually large in microbump bonding which can affect the TSV speed, energy efficiency, and cooling of the stack.

Potential directions: There are alternative bonding techniques that eliminate the need of solder interconnect (microbump) and hence may help overcome the issues related to the microbump bonding. Two potential alternative bonding techniques are hybrid bonding and direct oxide bonding. Neither of these techniques have been used yet for high-volume production of DRAM stacks.

Hybrid bonding [25, 26] using an annealing process to first bond the dielectric layers of two connecting dies and then subsequent bonding of the metal pads embedded in these dielectric layers through thermal expansion and diffusion. The

dielectric bonding is usually achieved at about 150^{0}C and the subsequent thermal bonding between the metal pads may require an annealing temperature of up to 400^{0}C. The hybrid bonding does not require microbumps and underfill which may lead to decreased stacking cost and complexity. It can also achieve small TSV pitch sizes and enable stacking of thin dies. However, the high annealing temperature for about an hour required for achieving thermal bonding of metal pads may degrade DRAM cell reliability. Hence, the feasibility of low-temperature hybrid bonding can be evaluated for stacked memories.

With direct oxide bonding [25], the surfaces of the stacked dies achieve oxide-to-oxide bonding using a low-temperature bonding process. Oxide bonding does not require physical modification of the CMOS process. However, TSVs are etched and formed after the bonding process. Since both diameter and length of TSVs in oxide bonding are smaller than those in microbump bonding (see Table 1), oxide bonding enables TSV connections with higher speed and better energy efficiency. The feasibility and cost-benefits of oxide bonding can be evaluated as well.

Summary: Bonding techniques have direct impacts on 3D die attributes such as TSV data rate, TSV density, stack height, reliability, and thermal conductivity. The existing bonding technique used in volume productions of HBM devices impose limitations on scaling the number of stacked DRAM dies. Investigating alternative bonding techniques for high-volume DRAM stacks is of paramount importance for realization of high-capacity DRAM stacks. Alternative bonding techniques can be a key enabler of memory stacks with high capacity, high bandwidth, and acceptable thermal conductivity.

5. Conclusions and Outlook

In this paper, we have provided an overview of major technical challenges in the design and implementation of high-capacity DRAM stacks that integrate many DRAM dies within a stack. We also presented potential directions to be investigated. We showed that the main obstacle facing high-capacity DRAM stacks is stacking a high number of DRAM dies to provide the required capacity while achieving high memory bandwidth and high thermal conductivity using a high-volume, high-yield production process. We believe that an alternative bonding and stacking technology would enable tall DRAM stacks with higher internal TSV interconnect density, less TSV capacitive load, better thermal conductivity within the stack, etc.

We hope that this paper sets the stage for more research and engineering of design and implementation of high-capacity, high-bandwidth DRAM stacks. As explained, future memory stacks present intriguing challenges, and thus research opportunities. We urge researchers from different domains such as packaging, bonding, reliability, and design architecture to participate in studying these challenges, investigating potential directions, and evaluating techniques to understand trade-offs and implications. Research in these domains would help address the challenges and pave the way for volume production of high-capacity, high-bandwidth memory stacks.

ACKNOWLEDGMENT

The authors would like to thank Rahul Agarwal, Milind Bhagavat, Bryan Black, Hayden Lee, Joe Macri, Aaron Nygren, Priyal Shah, and Vilas Sridharan for their valuable comments and suggestions throughout the course of this work. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] "High bandwidth memory (HBM) DRAM JESD235A," 2015. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235a

[2] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hot Chips*, 2011.

[3] "Addendum no. 1 to JESD79-4, 3D Stacked DRAM JESD79-4-1," 2017. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-4-1

[4] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, "Design and analysis of an APU for exascale computing," in *HPCA*, 2017, pp. 85–96.

[5] C. C. Lee, C. Hung, C. Cheung, P. F. Yang, C. L. Kao, D. L. Chen, M. K. Shih, C. L. C. Chien, Y. H. Hsiao, L. C. Chen, M. Su, M. Alfano, J. Siegel, J. Din, and B. Black, "An overview of the development of a GPU with integrated HBM on silicon interposer," in *IEEE Electronic Components and Technology Conference (ECTC)*, 2016, pp. 1439–1444.

[6] "AMD Radeon RX Vega^{64}," 2018. [Online]. Available: https://www.amd.com/en/products/graphics/radeon-rx-vega-64

[7] K. Gibb, "Hats off to Hynix: Inside 1st high bandwidth memory." [Online]. Available: https://www.eetimes.com/author.asp?section_id=36&doc_id=1327254

[8] D. Yudanov and M. Ignatowski, "Method and apparatus of integrating memory stacks," 2017, US Patent Application.

[9] C. Kim, H.-W. Lee, and J. Song, *High-Bandwidth Memory Interface*, 1st ed. Springer International Publishing, 2014.

[10] D. U. Lee, K. S. Lee, Y. Lee, K. W. Kim, J. H. Kang, J. Lee, and J. H. Chun, "Design considerations of HBM stacked DRAM and the memory architecture extension," in *IEEE Custom Integrated Circuits Conference (CICC)*, 2015, pp. 1–8.

[11] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous multi-layer access: Improving 3D-stacked memory bandwidth at low cost," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 63:1–63:29, 2016.

[12] S. B. Lim, H. W. Lee, J. Song, and C. Kim, "A 247 uW 800 Mb/s/pin DLL-based data self-aligner for through silicon via (TSV) interface," *IEEE J. of Solid-State Circuits*, vol. 48, no. 3, pp. 711–723, 2013.

[13] K. Sohn, W. J. Yun, R. Oh, C. S. Oh, S. Y. Seo, M. S. Park, D. H. Shin, W. C. Jung, S. H. Shin, J. M. Ryu, H. S. Yu, J. H. Jung, H. Lee, S. Y. Kang, Y. S. Sohn, J. H. Choi, Y. C. Bae, S. J. Jang, and G. Jin, "A 1.2 v 20 nm 307 GB/s HBM DRAM with at-speed wafer-level IO test scheme and adaptive refresh considering temperature distribution," *IEEE J. of Solid-State Circuits*, vol. 52, no. 1, pp. 250–260, 2017.

[14] "Technical note, uprating semiconductors for high-temperature applications," 2004. [Online]. Available: https://www.micron.com/~/media/documents/products/technical-note/dram/tn0018.pdf

[15] J. Kim and Y. Kim, "HBM: Memory solution for bandwidth-hungry processors," in *Hot Chips*, 2014.

[16] A. Agrawal, J. Torrellas, and S. Idgunji, "Xylem: Enhancing vertical thermal conduction in 3D processor-memory stacks," in *Micro*, 2017, pp. 546–559.

[17] J. Xie and M. Swaminathan, "Electrical-thermal co-simulation of 3d integrated systems with micro-fluidic cooling and joule heating effects," *IEEE Trans. on Components, Packaging and Manufacturing Technology*, vol. 1, no. 2, pp. 234–246, 2011.

[18] J. Cong, G. Luo, J. Wei, and Y. Zhang, "Thermal-aware 3D IC placement via transformation," in *Asia and South Pacific Design Automation Conference*, 2007, pp. 780–785.

[19] M. J. Khurshid and M. Lipasti, "Data compression for thermal mitigation in the hybrid memory cube," in *Intl. Conf. on Computer Design (ICCD)*, 2013, pp. 185–192.

[20] H. Jeon, G. H. Loh, and M. Annavaram, "Efficient RAS support for die-stacked DRAM," in *Intl. Test Conf.*, 2014, pp. 1–10.

[21] N. H. Khan, S. M. Alam, and S. Hassoun, "System-level comparison of power delivery design for 2D and 3D ICs," in *Intl. Conf. on 3D System Integration*, 2009, pp. 1–7.

[22] M. Shevgoor, J. S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. N. Udipi, "Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device," in *Micro*, 2013, pp. 198–209.

[23] U. Kang, H.-J. Chung, S. Heo, S.-H. Ahn, H. Lee, S.-H. Cha, J. Ahn, D. Kwon, J. H. Kim, J.-W. Lee, H.-S. Joo, W.-S. Kim, H.-K. Kim, E.-M. Lee, S.-R. Kim, K.-H. Ma, D.-H. Jang, N.-S. Kim, M.-S. Choi, S.-J. Oh, J.-B. Lee, T.-K. Jung, J.-H. Yoo, and C. Kim, "8Gb 3D DDR3 DRAM using through-silicon-via technology," in *Intl. Solid-State Circuits Conference - Digest of Technical Papers*, 2009, pp. 130–131,131a.

[24] T. Nomura, R. Mori, K. Takayanagi, K. Fukuoka, and K. Nii, "Design challenges in 3-D SoC stacked with a 12.8 GB/s TSV wide I/O DRAM," *IEEE J. on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 3, pp. 364–372, 2016.

[25] L. Di Cioccio, P. Gueguen, R. Taibi, D. Landru, G. Gaudin, C. Chappaz, F. Rieutord, F. de Crecy, I. Radu, L. L. Chapelon, and L. Clavelier, "An overview of patterned metal/dielectric surface bonding: Mechanism, alignment and characterization," *J. of The Electrochemical Society*, vol. 158, no. 6, pp. P81–P86, 2011.

[26] G. Gao, L. Mirkarimi, G. Fountain, L. Wang, C. Uzoh, T. Workman, G. Guevara, C. Mandalapu, B. Lee, and R. Katkar, "Scaling package interconnects below 20Âµm pitch with hybrid bonding," in *IEEE Electronic Components and Technology Conf. (ECTC)*, 2018, pp. 314–322.

Evaluation of Intel Memory Drive Technology Performance for Scientific Applications

Vladimir Mironov
Department of Chemistry,
Lomonosov Moscow State University
Moscow, Russian Federation
vmironov@lcc.chem.msu.ru

Andrey Kudryavtsev
Intel Corporation
Folsom, California, USA
andrey.o.kudryavtsev@intel.com

Yuri Alexeev
Argonne National Laboratory,
Computational Science Division
Argonne, Illinois, USA
yuri@alcf.anl.gov

Alexander Moskovsky
RSC Technologies
Moscow, Russian Federation
moskov@rsc-tech.ru

Igor Kulikov
Institute of Computational
Mathematics and Mathematical
Geophysics SB RAS
Novosibirsk, Russian Federation
kulikov@ssd.sscc.ru

Igor Chernykh
Institute of Computational
Mathematics and Mathematical
Geophysics SB RAS
Novosibirsk, Russian Federation
chernykh@ssd.sscc.ru

ABSTRACT

In this paper, we present benchmark data for Intel Memory Drive Technology (IMDT), which is a new generation of Software-defined Memory (SDM) based on Intel ScaleMP collaboration and using 3D XPoint ™ based Intel Solid-State Drives (SSDs) called Optane. We studied IMDT performance for synthetic benchmarks, scientific kernels, and applications. We chose these benchmarks to represent different patterns for computation and accessing data on disks and memory. To put performance of IMDT in comparison, we used two memory configurations: hybrid IMDT DDR4/Optane and DDR4 only systems. The performance was measured as a percentage of used memory and analyzed in detail. We found that for some applications DDR4/Optane hybrid configuration outperforms DDR4 setup by up to 20%.

CCS CONCEPTS

• **Hardware** → **Non-volatile memory**; • **Computing methodologies** → *Massively parallel and high-performance simulations*; • **Applied computing**; • **Software and its engineering** → *Memory management*;

KEYWORDS

Intel Memory Drive Technology, Solid State Drives, Intel Optane, ScaleMP

ACM Reference Format:
Vladimir Mironov, Andrey Kudryavtsev, Yuri Alexeev, Alexander Moskovsky, Igor Kulikov, and Igor Chernykh. 2018. Evaluation of Intel Memory Drive Technology Performance for Scientific Applications. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3286475.3286479

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286479

1 INTRODUCTION

In the recent years the capacity of system memory for high performance computing (HPC) systems has not been kept with the pace of the increased central processing unit (CPU) power. The amount of system memory often limits the size of problems that can be solved. System memory is typically based on dynamic random access memory (DRAM). DRAM prices have significantly grown up in the recent year. In 2017, DRAM prices were growing up approximately 10-20% quarterly [10]. As a result, memory can contribute up to 90% to the cost of the servers.

A modern memory system is a hierarchy of storage devices with different capacities, costs, latencies, and bandwidths intended to reduce price of the system. It makes a perfect sense to introduce yet another level in the memory hierarchy between DRAM and hard disks to drive price of the system down. Solid-State Drives (SSDs) are a good candidate because they are cheaper than DRAM up to 10 times. What is more important, over the last 10 years, SSDs based on NAND technology emerged with higher read/write speed and Input/Ouput Operations per Second (IOPS) metric than hard disks.

Recently, Intel announced [21] a new SSD product based on novel 3D XPoint™ technology under the name Intel® Optane™. It was developed to overcome the drawbacks of NAND-technology: block-based memory addressing and limited write endurance. To be more specific, with 3D XPoint each memory cell can be addressed individually and write endurance of 3D XPoint memory is significantly higher than NAND SSDs. As a result, 3D XPoint flash memory can be used instead of DRAM, albeit as a slow memory, which can be still an attractive solution given that Intel Optane is notably cheaper than random access memory (RAM) per gigabyte. A novel Intel Memory Drive Technology (IMDT) allows to use Intel Optane drives as a system memory. Another important advantage of 3D XPoint compared to DRAM is that it has a high density of memory cells, which allows to build compact systems with massive memory banks.

In this work, we evaluated the capabilities of Intel Optane drives together with IMDT for numerical simulations requiring large amount of memory. We started with the overview of IMDT technology in section 2. In section 3, we described the methodology. In

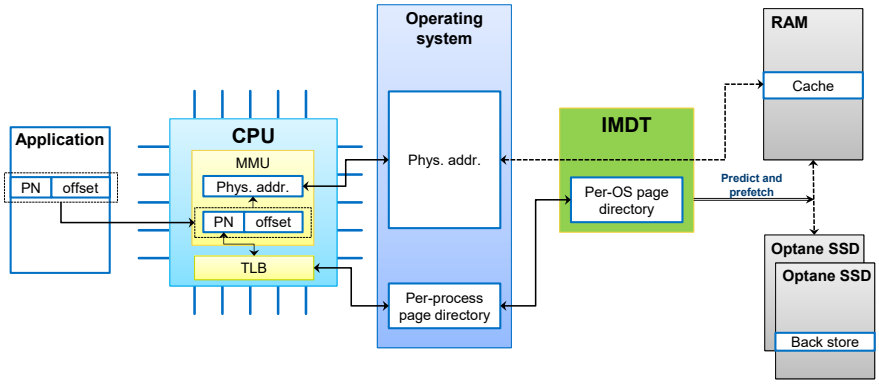


Figure 1: This figure describes how Intel Memory Drive Technology works. Solid lines represent inquiry, dashed lines represent data transfer, and double lines represent commands issued.

Sections 4 and 5 we described all benchmarks and corresponding performance results. In section 6 we discussed the performance results, and in Section 7 we presented our conclusions and plans for the future work.

2 OVERVIEW OF INTEL MEMORY DRIVE TECHNOLOGY

For effective use of Intel Optane in hybrid RAM-SSD memory systems, Intel corporation and ScaleMP developed a technology called IMDT [4, 7]. IMDT integrates the Intel Optane into the memory subsystem and makes it appear like RAM to the operating system and applications. Moreover, IMDT increases memory capacity beyond RAM limitations and performs in a completely transparent manner without any changes in operating system and applications. The key feature of IMDT is that RAM is effectively used as cache. As a result, IMDT can achieve good performance compared to all-RAM systems for some applications at a fraction of the cost as we have shown in this paper.

ScaleMP initially has developed a technology to make virtual non-uniform memory access (NUMA) system using high speed node interconnect of modern high performance computational clusters. NUMA systems are typically defined as any contemporary multi-socket system. It allows a processor to access memory at varying degrees of latency or "distance" (e.g. memory attached to another processor), over a network or fabric. In some cases, this fabric is purpose-built for such processor communication, like Intel® QuickPath and UltraPath Interconnects (Intel® QPI and UPI respectively). In other cases, standard fabrics such as Peripheral Component Interconnect Express (PCIe) or Intel® Omni-Path Fabric are used for the same purpose along with software-defined memory (SDM) to provide memory coherency, operating as if additional memory was installed in the system.

Accessing memory at varying lower performance over networks has proven to be feasible and useful by using predictive memory access technologies that support advanced caching and replication, effectively trading latency for bandwidth. This is exactly what IMDT is doing to enable non-volatile memory (NVM) to be used as system memory. Instead of doing it over fabric, however, it does so with storage. With IMDT, most of the Intel Optane capacity is transparently used as an extension to the DRAM capacity of the system.

IMDT is implemented as an operating system (OS)-transparent virtual machine (Figure 1). In IMDT, Intel Optane SSDs are used as part of the system memory to present the aggregated capacity of the DRAM and NVM installed in the system as one coherent, shared memory address space. No changes are required to the operating system, applications, or any other system components. Additionally IMDT implements advanced memory access prediction algorithms to optimize memory access performance.

A popular approach to virtualize disk memory is to store part of virtual memory (VM) pages on special disk partition or file is implemented in all popular operating systems nowadays. However, the resulting performance is very sensitive not only to the storage speed but also to VM manager implementation. It is very important to correctly predict which memory page on disk will be needed soon and to load it in RAM to avoid program spinning in a page fault state. The built-in OS swap in Linux kernel is not very intelligent and usually affected by this problem. On the contrary, IMDT analyzes memory access patterns and prefetches the data into the RAM "cache" (Figure 1) before it is used, resulting in better performance.

IMDT leverage the low-latency media access provide by Intel Optane SSDs. NAND SSD latency cannot be improved by simply aggregating multiple drives. Transitioning to Intel Optane SSDs is another step forward to the reductions of the gap between DRAM and SSD performance by using lower latency media based on the 3D XPoint technology. However, DRAM still has lower latency than Intel Optane, which can potentially affect the performance of applications with DRAM+Optane configuration studied in this paper.

3 METHODOLOGY

IMDT architecture is based on the hypervisor layer which manages paging exclusively. This makes a hybrid memory transparent from one side, however, standard CPU counters become unavailable to performance profiling tools. Thus, we took an approach to make a comparison with DRAM-based system side by side. The efficiency metric was calculated as a ratio of software defined performance

counters, if available, or simply the ratio of the time-to-solution on DRAM-based system and IMDT-based system.

3.1 Hardware and software configuration

In this study, we used dual-socket Intel Broadwell (Xeon E5 2699 v4, 22 cores, 2.2 GHz) node with latest version of BIOS. We have used two memory configurations for this node. In the first configuration, it was equipped with 256 GB DDR4 registered ECC memory (16×16 GB Kingston 2133 MHz DDR4) and four Intel® Optane™ SSDs P4800X (320 GB memory mode). We used Intel Memory Drive Technology 8.2 to expand system memory with Intel SSDs up to approximately 1,500 GB. In the second configuration, the node was exclusively equipped by 1,536 GB of DDR4 registered ECC memory (24×64 GB Micron 2666 MHz DDR4). In both configurations we used a stripe of four 400 GB Intel DC P3700 SSD drives as a local storage. Intel Parallel Studio XE 2017 (update 4) was used to compile the code for all benchmarks. Hardware counters on non-IMDT setup were collected using Intel® Performance Counter Monitor [5].

3.2 Data size representation

IMDT assumes that all data is loaded in the RAM before it is actually used. It is important to note that if the whole dataset fits in the RAM, it is very unlikely that it will be moved to the Optane disks. In this case, the difference between IMDT and RAM should be negligible. The difference will be more visible only when the data size is significantly larger than the available RAM. Since the performance results are connected to the actual RAM size, we find more convenient to represent benchmark sizes in parts of RAM in IMDT configuration (256 GB) and not in GB or problem dimensions. Such representation of data sets is more general and the results can be extrapolated to different hardware configurations.

4 DESCRIPTION OF BENCHMARKS

In this section, we described various types of benchmarks to evaluate performance of IMDT. We broadly divided benchmarks in three classes – synthetic benchmarks, scientific kernels, and scientific applications. The goal is to test performance for a diverse set of scientific applications, which have different memory access patterns with various memory bandwidth and latency requirements.

4.1 Synthetic benchmarks

4.1.1 STREAM. [17] is a simple benchmark commonly used to measure sustainable bandwidth of the system memory and corresponding computation rate for a few simple vector kernels. In this work, we have used multi-threaded implementation of this benchmark. We studied memory bandwidth for a test requiring ≈ 500 GB memory allocation on 22, 44, and 88 threads.

4.1.2 Polynomial benchmark. was used to compute polynomials of various degree of complexity. Polynomials are commonly used in mathematical libraries for fast and precise evaluation of various special functions. Thus, they are virtually present in all scientific programs. In our tests, we calculated polynomials of predefined degree over a large array of double precision data stored in memory.

Memory access pattern is similar to the STREAM benchmark. The only difference that we can finely tune the arithmetic intensity of the benchmark by changing the degree of computed polynomials. From this point of view STREAM benchmark is a particular case of the polynomial benchmark when the polynomial degree is zero (*STREAM copy*) or one (*STREAM scale*). We used Horner's method of polynomial evaluation which is efficiently translated to the fused multiply-add (FMA) operations.

We have calculated performance for polynomials of degrees 16, 64, 128, 256, 512, 768, 1024, 2048, and 8192 using various data sizes (from 50 to 900 GB). We studied two data access patterns. In the first one we just read the value from the array of arguments, calculate the polynomial value and add it to a thread-local variable. There is only one (*read*) data stream to the IMDT disk storage in this case. In another case the result of polynomial calculation updates corresponding value in the array of arguments. There are two data streams here (*read* and *write*). Arithmetic intensity of this benchmark was calculated as follows:

$$AI = 2 \cdot \frac{polynomial\ degree}{sizeof(double)}, \tag{1}$$

where factor two corresponds to the one addition and one multiplication for each polynomial degree in Horner's method of polynomial evaluation.

4.1.3 GEMM. (GEneral Matrix Multiplication) is one of the core routines in Basic Linear Algebra Subprograms (BLAS) library. It is a level 3 BLAS operation defining matrix-matrix operation. GEMM is often used for performance evaluations and it is our first benchmark to evaluate IMDT performance. GEMM is a compute-bound operation with $O(N^3)$ arithmetic operations and $O(N^2)$ memory operations, where N is a leading dimension of matrices. Arithmetic intensity grows as $O(N)$ depending on matrix size and is flexible. The source code of the benchmark used in our tests is available here [6].

4.2 Scientific kernels

4.2.1 LU decomposition. (where "LU" stands for "lower upper" of a matrix, and also called LU factorization) is a commonly used kernel in a number of important linear algebraic problems like solving system of linear equations, finding eigenvalues, etc. In current study, we used Intel Math Kernel Library (MKL) [3] implementations of LU decomposition, more specifically `dgetrf` and `mkl_dgetrfnpi`. We also studied the performance of an LU decomposition algorithm using tile algorithm, which dramatically improved performance of IMDT. The source code of the latter was taken from the *hetero-streams* code base [2].

4.2.2 Fast Fourier Transform (FFT). is an algorithm that samples a signal over a period of time or space and divides it into its frequency components. FFT is an important kernel in many scientific codes. In this work, we have studied the performance of the FFT implemented in MKL library [3]. We have used three-dimensional decomposition of the $N \times N \times N$ grid data. The benchmark sizes were $N = (500 \div 5800)$ resulting in 0.001-1.5 TB memory footprint.

4.3 Scientific applications

4.3.1 LAMMPS. (Large-scale Atomic/Molecular Massively Parallel Simulator) is a popular molecular simulation package developed in Sandia National Laboratory [18]. Its main focus is a force-field based

molecular dynamics. We have used scaled Rhodopsin benchmark distributed with the source code. Benchmark set was generated from the original chemical system (32,000 atoms) by its periodical replication in X, Y (8 times) and Z (8-160 times) dimensions. The largest chemical system comprises 328,000,000 atoms ($\approx$ 1.1 TB memory footprint). Performance metrics – number of molecular dynamics steps per second.

4.3.2 GAMESS. (General Atomic and Molecular Electronic Structure System) is one of the most popular quantum chemistry packages. It is a general purpose program, where a large number of quantum chemistry methods are implemented. We used the latest version of code distributed from GAMESS website [8]. In this work, we have studied the performance of the Hartree-Fock method. We have used stacks of benzene molecules as a model chemical system. By changing the number of benzene molecules in stack we can vary memory footprint of the application. 6-31G(d) basis set was used in the simulations.

4.3.3 AstroPhi. is a hyperbolic PDE engine which is used for numerical simulation of astrophysical problems [1]. AstroPhi realizing a multi-component hydrodynamic model for astrophysical objects interaction. The numerical method of solving hydrodynamic equations is based on a combination of an operator splitting approach, Godunov's method with modification of Roe's averaging, and a piecewise-parabolic method on a local stencil [19, 20]. The redefined system of equations is used to guarantee the non-decrease of entropy [15] and for speed corrections [11]. The detailed description of a numerical method can be found in [16]. In this work, we used the numerical simulation of gas expansion into vacuum for benchmarking. We have used 3D arrays with up to 2000^3 size ($\approx$ 1.5 TB memory footprint) for this benchmark.

4.3.4 PARDISO. is a package for sparse linear algebra calculations. It is a part of Intel MKL library [3]. In our work, we studied the performance of the Cholesky decomposition of sparse ($O(N)$ nonzero elements) $N \times N$ matrices, where $N = (5, 10, 20, 25, 30, 35, 40) \cdot 10^6$. Memory footprint of benchmarks varied from 36 to 790 GB.

4.3.5 Intel-QS. (former qHiPSTER) is a distributed high-performance implementation of a quantum simulator on a classical computer, that can simulate general single-qubit gates and two-qubit controlled gates [22]. The code is fully parallelized with MPI and OpenMP. The code is architectured in the way that memory consumption exponentially grows as more qubits are being simulated. We benchmarked a provided quantum FFT test for 30-35 qubit simulations. 35 qubits simulation required more than 1.5TB of memory. The code used in our benchmarks was taken from Intel-QS repository on Github [9].

5 RESULTS

5.1 Synthetic benchmarks

5.1.1 STREAM. benchmark was used as a reference to a worst case scenario, where application has low CPU utilization and high memory bandwidth requirements. We obtained 80 GB/s memory bandwidth for the DRAM-configured node, while for IMDT-configured

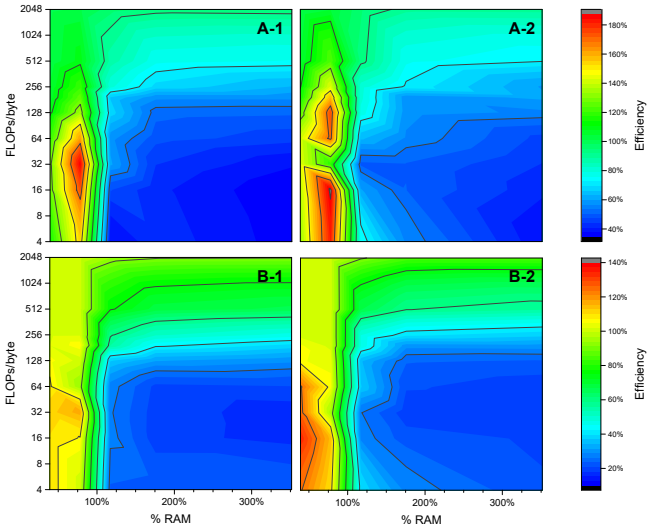


Figure 2: Polynomial benchmark results. (A) – one data stream for 44 (A-1) and 88 (A-2) threads. (B) – two data streams for 44 (B-1) and 88 (B-2) threads. The efficiency is denoted by color with a legend on corresponding row. See text for more details.

node we got 10 GB/s memory bandwidth for the benchmarks requesting maximum available memory. In other words, we are comparing best case scenario for DRAM bandwidth with the worst case scenario on IMDT. Thus, we can expect the worst possible efficiency of 10/80 = 12.5% IMDT vs DRAM. It should be noted that running benchmarks which fit in DRAM cache of IMDT results in the bandwidth equal to (80-100 GB/s), which is comparable to the DRAM bandwidth. This is what we expected and it is the proof that IMDT utilizes optimally DRAM cache. The measured bandwidth actually depends only on the number of threads and it was higher for the less concurrent jobs. It applies only to IMDT benchmarks requesting memory smaller than the size of DRAM cache. Memory bandwidth of DRAM-configured node does not depend on the workload size nor on the number of threads.

5.1.2 Polynomial benchmark. Results of the polynomial benchmarks are presented in Figure 2. As one can see in Figure 2, patterns of efficiency are very similar. If the data fits in the RAM cache of IMDT then IMDT typically shows better performance than DRAM-configured node, especially for short polynomials. High concurrency (88 threads, Figure 2 (A-2 and B-2)) is also beneficial to IMDT in these benchmarks. However, a better efficiency can be obtained for benchmarks with higher order of polynomials. In terms of arithmetic intensity (eq. (1)), it is required to have at least 256 floating-point operations (FLOPs) per byte to get IMDT efficiency close to 100%. It will be discussed later in detail (see Section 5.4).

5.1.3 GEMM benchmark. According to our benchmarks shown in Figure 3, GEMM shows very good efficiency for every problem size. All observed efficiencies vary from 90% for large benchmarks to 125% for small benchmarks. Such efficiency is expected because GEMM is purely compute bound. To be more specific, the arithmetic

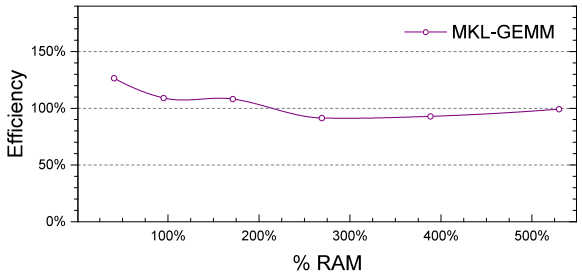


Figure 3: IMDT efficiency plot for GEMM benchmark. Higher efficiency is better. 100% efficiency corresponds to DRAM performance.

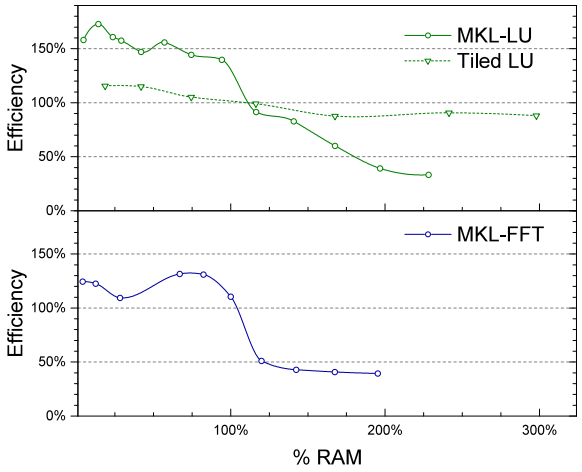


Figure 4: IMDT efficiency plots for LU and FFT benchmarks. Two implementations (MKL and tiled) of LU decomposition were benchmarked. Higher efficiency is better. 100% efficiency corresponds to DRAM performance.

intensity even for a relatively small GEMM benchmark is much higher than the required value of 250 FLOP/byte per data stream, which was estimated in polynomial benchmarks. In our tests, we have used custom ("segmented") GEMM benchmark with improved data locality: all matrices are stored in a tiled format (arrays of "segments") and matrix multiplication goes tile-by-tile. Thus the arithmetic intensity of all benchmarks is constant and equal to the arithmetic intensity of a simple tile-by-tile matrix multiplication. It is approximately equal to 2 FLOPs multiplied by tile dimension and divided by the size of data type in bytes (4 for `float` and 8 for `double`). In our benchmark with a single-precision GEMM with typical tile dimension size of ≈43000, the arithmetic intensity is ≈21500 FLOPs/byte, which is far beyond the required 250 FLOPs/byte.

5.2 Scientific kernels

5.2.1 LU decomposition. The efficiency of LU decomposition implemented in MKL library strongly depends on the problem size. The efficiency is excellent when a matrix fits into the memory (Figure 4). We observe about 150% – 180% speedup on IMDT for small matrices. However, the efficiency decreases down to ≈ 30% for very large matrices with leading dimension equal or greater than $\approx 2 \cdot 10^5$. This result was unexpected; in fact, our LU implementation calls BLAS level 3 functions such as GEMM, which has excellent efficiency on IMDT as we demonstrated in previous section. We can provide two explanations for unfavorable memory access patterns in LU decomposition. First one is the partial pivoting which interchanges rows and columns in the original matrix. Second, matrix is stored as a contiguous array in memory that is known for its inefficient memory access to the elements of the neighboring columns (rows in case of Fortran). Both problems are absent in a special tile-based LU decomposition benchmark implemented in *hetero-streams* code base. We also ran benchmarks for this optimized LU decomposition benchmark. Tiling of the matrix not only improved the performance by about 20% for both "DRAM" and "IMDT" memory configurations, but also improved the efficiency of IMDT to ≈90% (see Figure 4). Removing of pivoting only without introducing matrix tiling does not significantly improve the efficiency of LU decomposition.

5.2.2 Fast Fourier Transform. The results of MKL-FFT benchmark are similar to those obtained for MKL-LU as shown in Figure 4. For small problem sizes the efficiency of IMDT exceeds 100%, but for large benchmarks the efficiency drops down to ≈ 40%. Performance drop occurs at 100% of RAM utilization. FFT problems typically have relatively small arithmetic intensity (small ratio of FLOPs/byte). Thus, obtaining relatively low IMDT efficiency was expected. We still believe that the FFT benchmark can be optimized for memory locality to improve IMDT efficiency even higher (see [12–14] for the examples of memory-optimized FFT implementations).

5.3 Scientific applications

The benchmarking results for different scientific applications are shown in Figure 5 and Figure 6. The applications are PARDISO, AstroPhi, LAMMPS, Intel-QS, and GAMESS. All applications except PARDISO show similar efficiency trends. When a benchmark requests memory smaller than the amount of available DRAM, the application performance on the IMDT-configured node is typically higher than for DRAM-configured node. At a certain threshold, which is typically a multiple of DRAM size, the IMDT efficiency declines based on the CPU Flop/S and memory bandwidth requirements.

5.3.1 MKL-PARDISO. PARDISO is very different from other studied benchmarks. The observed IMDT efficiency is 120-140% of "DRAM"-configured node for all studied problem sizes. It was not very surprising because Cholesky decomposition is known to be compute intensive. MKL-PARDISO is optimized for the disk-based out-of-core calculations resulting in excellent memory locality to access data structures. As a result, this benchmark always benefits from faster access to the non-local NUMA memory on IMDT which results in the improved performance on "IMDT"-configured node.

5.3.2 AstroPhi. In Figure 5 we presented the efficiency plot of Lagrangian step of the gas dynamic simulation, which is the most time consuming step (> 90% compute time). This step describes the convective transport of the gas quantities with the scheme velocity for the gas expansion into vacuum problem. The number

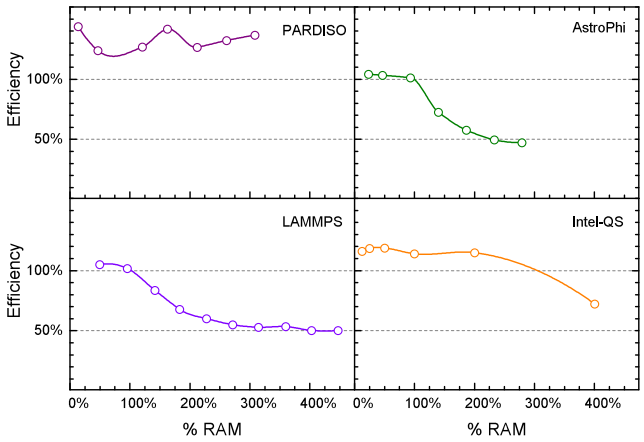


Figure 5: IMDT efficiency plots for various scientific applications. Higher efficiency is better. 100% efficiency corresponds to DRAM performance.

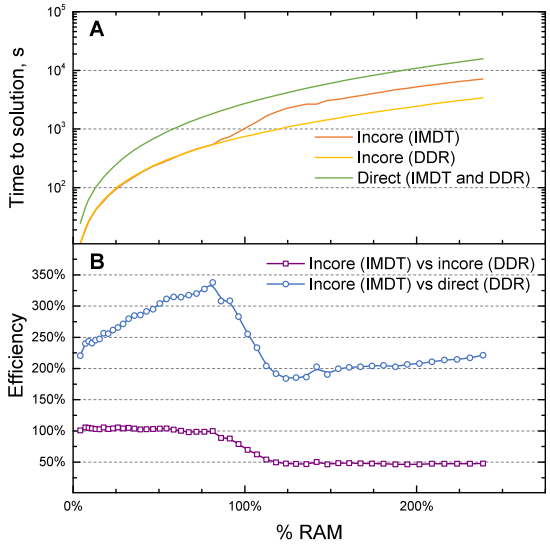


Figure 6: GAMESS Hartree-Fock simulation (10 iterations) for stacks of benzene molecules with 6-31G(d) basis set. (A) Time to solution in seconds, lower is better. (B) IMDT efficiency, higher efficiency is better, 100% efficiency corresponds to DRAM performance. The performance of the direct Hartree-Fock on "IMDT" and "DRAM"-configured node (see text for details) is the same ((A), green line).

of FLOPs/byte is not very high and the efficiency plot follows the trend we described above. We observe a slow decrease in efficiency down to ≈50% when the data does not fit into DRAM cache of IMDT. Otherwise the efficiency is close to 100%.

5.3.3 LAMMPS. We studied the performance of the molecular dynamics of the Rhodopsin benchmark provided with LAMMPS distribution. It is an all-atom simulation of the solvated lipid bilayer surrounding Rhodopsin protein. The calculations are dominated by the long-range electrostatics interactions in particle-particle mesh algorithm. The results of benchmarks are presented in Figure 5. It is obvious that LAMMPS efficiency follows the same pattern as AstroPhi and FFT: it is more than 100% when tests fit in the DRAM cache of the IMDT and it is dropping down when tests do not fit. For tests with high memory usage the efficiency is ≈50%.

5.3.4 Intel-QS. We benchmarked Intel-QS using provided quantum FFT example for 30-35 qubits. Actually, each additional qubit doubles the amount of memory required for the job. For that reason we had to stop at 35 qubit test which occupy about 1 TB of memory. The observed IMDT efficiency was greater than 100% for 30-34 qubits and drops down to ≈70% at 35 qubit simulation. A significant portion of the simulation take FFT steps. Thus, degradation of the performance at high memory utilization was not surprising. However, the overall efficiency is almost two times better than for FFT benchmark.

5.3.5 GAMESS. We studied the performance of the two Hartree-Fock method (HF) algorithms. HF is solved iteratively and for each iteration a large number of electron-repulsion integrals (ERIs) need to be re-computed (direct HF) or read from disk or memory (conventional HF). In the special case of conventional HF called incore HF, ERIs are computed once before HF iterations and stored in DRAM. In the subsequent HF iterations, the computed ERIs are read from memory. We benchmarked both direct and incore HF methods. The former algorithm has small memory footprint, but re-computation of all ERIs each iteration (typical number of iterations is 20) results in much longer time to solution compared to incore HF method if ERIs fit in memory.

The performance of the direct HF method on "DRAM" and "IMDT"-configured nodes is very similar (see Figure 6 (A), green line). However, the performance of the incore method differs between "DRAM" and "IMDT" (see Figure 6 (A), red and yellow lines). The efficiency shown in Figure 6 (B) for incore IMDT vs incore DRAM (purple line) behaves similar to other benchmarks – when benchmarks fits in the DRAM cache of the IMDT then the efficiency is close to 100%, otherwise it decreases to ≈50%. But for incore IMDT vs direct DRAM (blue line) the efficiency is much better. The efficiency varies between approximately 200% and 350%. Thus, IMDT can be used to speed up Hartree-Fock calculations when the amount of DRAM is not available to fit all ERIs in memory.

5.4 Analysis of IMDT performance

Modern processors can overlap data transfer and computation very efficiently. A good representative example is the Polynomial benchmark (see Section 4.1.2). When the polynomial degree is low the time required to move data from system memory to CPU is much higher than the time of polynomial computation (Section 5.1.2, Figure 2). In this case, the performance is bound by memory bandwidth. By increasing the amount of computation, the overlap between data transfer and computation becomes more efficient and the benchmark gradually transforms from a memory-bound to a compute-bound problem. Increasing of arithmetic intensity is achieved by increasing the degree of polynomials (see eq. (1)).

On Figure 7 the dependence of average DRAM bandwidths on the arithmetic intensity is shown for polynomial benchmarks with

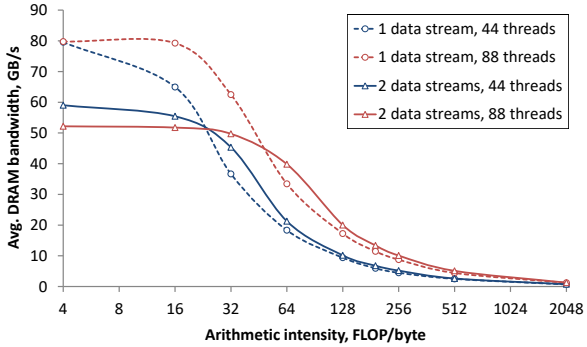


Figure 7: Average DRAM bandwidth in the polynomial benchmark depending on the arithmetic intensity (polynomial degree), number of data streams (1 data stream – read-only access pattern, 2 data streams – read/write access pattern, see text for details), and the number of working threads. The results were obtained by Intel Processor Counter Monitor (sum of System READ and WRITE counters).

different number of data streams. The improvement of overlap between data transfer and computation is observed at about 16-32 FLOP/byte. The computation of low-order polynomials (less than 16 FLOP/byte) is not limited by the compute power, resulting in high memory bandwidth. The bandwidth value depends on I/O pattern (i.e. the number of data streams) and it is limited by the DRAM memory bandwidth, which is about 80 GB/s.

The bandwidth dependence on the number of data streams of low-order polynomials results from NUMA memory access. If the benchmark was optimized for NUMA, the highest bandwidth for one and two data streams would be the same. However, in IMDT architecture, while application thread accesses the remote NUMA node for writes, IMDT places the data to the DRAM attached to the local NUMA node. This can significantly reduce the pressure on the cross-socket link and as a result the performance can become better than DRAM based system performance. It is exactly what we observed in our tests when all the data fits in the DRAM cache (see Figure 2).

When the arithmetic intensity grows beyond 16 FLOP/byte, the memory bandwidth starts decreasing. At 64 FLOP/byte and beyond the benchmark becomes compute bound. It means that the memory bandwidth does not depend on the number of data streams but on the availability of computational resources (i.e. number of threads). However, the memory bandwidth decreases slowly with the arithmetic intensity (Figure 7). Taking into account that the memory bandwidth of our IMDT system is capped by 10 GB/s, we expect that only those benchmarks that are below this threshold will have good efficiency. It is expected that it will apply to all benchmarks with different problem sizes. In terms of the arithmetic intensity, it corresponds to $\approx 128 - 256$ FLOP/byte. This correlation is shown on Figure 2. The same analysis can be applied to any benchmark to estimate the potential efficiency of the IMDT approach.

5.5 Summary

To sum up our benchmarking results, our tests show that there is virtually no difference between using DRAM and IMDT if a benchmark requires memory less than the amount of available RAM. IMDT correctly handles these cases, if the test fits in RAM and there is no need to use Optane memory. In fact, IMDT frequently outperforms RAM because IMDT has advanced memory management system. The situation is very different for large tests. For some tests like dense linear algebra, PARDISO and Intel-QS efficiency remains high, while for other applications like LAMMPS, AstroPhi, and GAMESS the efficiency slowly declines to about 50%. Even in the latter case IMDT can be attractive for scientific users since it enables larger problem sizes to be addressed.

6 DISCUSSION

One of the most important benefits of IMDT is that it significantly reduces data traffic trough Intel QuickPath Interconnect (QPI) bus on NUMA systems. For example, GEMM unoptimized benchmark on "DRAM"-configured node performs about 20-50% slower for large datasets compared to a small ones. The main reason is overloaded QPI bus. When a benchmark saturates QPI bandwidth then it causes CPU stalls waiting for data. QPI bandwidth in our system is 9.6 GT/s (1 GT/s= 10^9 transfers per second) or ≈10 GB/s unidirectional (≈20 GB/s bidirectional) and it can easily become a bottleneck. It is an inherent issue of multisocket NUMA-systems which adversely affects performance of not only GEMM, but any other applications.

There are a few ways to resolve this issue. For example, in optimized GEMM implementation [6] matrices are split to tiles, which are placed in memory intelligently taking into the account the "first-touch" memory allocation policy in Linux OS. As a result, QPI bus load drops to 5-10% and performance significantly improves achieving almost theoretical peak. It was observed in our experiments with GEMM by using Intel Performance Counter Monitor (PCM) software [5].

There is no such issue with IMDT and performance is consistently close to theoretical peak even for the unoptimized GEMM implementation. IMDT provides optimal access to the data on the remote NUMA node improving the efficiency of almost all applications. This is why we almost never seen in practice a very low IMDT efficiency even for strongly memory-bandwidth bound benchmarks like FFT and AstroPhi. Theoretical efficiency minimum of 12.5% was observed only for the specially designed synthetic benchmarks like STREAM and polynomial benchmark.

However, IMDT is not a solution to all memory-related issues. For example, it cannot help in situations when an application has random memory access patterns across a large number of memory pages with a low degree of application parallelism. While the performance penalty is not very high for DRAM memory, frequent access of the IMDT backstore on SSD can be limited by the bandwidth of Intel Optane SSD, and IMDT can only compensate for that if the workload has a high degree of parallel memory accesses (using many threads or many processes concurrently). In such cases, it may be beneficial to redesign data layout for better locality of data structures. In this work, we observed it when we ran MKL

implementation of LU decomposition. Switching to the tiled implementation of the LU algorithm results in the significantly improved efficiency of IMDT because of better data locality. The similar approach can be applied to other applications. However, it is beyond the scope of this paper and it is a subject for our future studies.

7 CONCLUSIONS AND FUTURE WORK

IMDT is a revolutionary technology that flattens the last levels of memory hierarchy: DRAM and disks. One of the major IMDT advantages is the high density of memory. It will be feasible in the near future to build systems with many terabytes of Optane memory. In fact, the bottleneck will not be the amount of Optane memory, but the amount of available DRAM cache for IMDT. It is currently possible to build an IMDT system with 24 TB of addressable memory (with 3 TB DRAM cache), which is not possible to build with DRAM. Even if it was possible to build a such system, IMDT offers a more cost effective solution.

There are HPC applications with large memory requirements. It is a common practice for such applications to store data in parallel network storage or use distributed memory. In this case, the application performance can be limited by network bandwidth. There is now another alternative which is to use DRAM+Optane configuration with IMDT. In theory, the bandwidth of the multiple striped Optane drives exceeds network bandwidth. IMDT especially benefits the applications that poorly scale on the multi-node environment due to high communication overhead. A good example of such application is a quantum simulator. Indeed, Intel-QS simulator efficiency shown in Figure 5 is excellent compared to other applications. Another good application that fits profile is the visualization of massive amount of data. We plan to explore the potential of IMDT for such applications in our future work.

IMDT prefetching subsystem analyzes memory access patterns in the real time and makes appropriate adjustments according to workload characteristics. This feature is crucial for IMDT performance and differentiates it from other solutions such as OS swap. We plan to analyze it in detail in our future work.

This work is important because we systematically studied performance of IMDT technology for a diverse set of scientific applications. We have demonstrated that applications and benchmarks exhibit reasonable performance level, when the system main memory is extended with the help of IMDT by Optane SSD. In some cases, we have seen DRAM+Optane configuration to outperform DRAM-only system by up to 20%. Based on performance analysis, we provide recipes how to unlock full potential of IMDT technology. It is our hope that this work will educate professionals about this new exciting technology and promote its wide-spread use.

8 ACKNOWLEDGEMENTS

This research used the resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy (DOE) Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. We thank the Intel® Parallel Computing Centers program for funding, ScaleMP team for technical support, RSC Group and Siberian Supercomputer Center ICMMG SB RAS for providing access to hardware, and Gennady Fedorov for help with Intel® PARDISO benchmark. This work was partially supported by the Russian Fund of Basic Researches grant 18-07-00757, 18-01-00166 and by the Grant of the Russian Science Foundation (project 18-11-00044).

REFERENCES

[1] 2018. AstroPhi. The hyperbolic PDE engine . https://github.com/IgorKulikov/AstroPhi
[2] 2018. Hetero Streams Library. https://github.com/01org/hetero-streams
[3] 2018. Intel Math Kernel Library. https://software.intel.com/mkl
[4] 2018. Intel Memory Drive Technology. https://www.intel.com/content/www/us/en/software/intel-memory-drive-technology.html
[5] 2018. Intel Performance Counter Monitor – A better way to measure CPU utilization. www.intel.com/software/pcm
[6] 2018. Segmented SGEMM benchmark for large memory systems. https://github.com/ScaleMP/SEG_SGEMM
[7] 2018. Software Defined Memory at a Fraction of the DRAM Cost – white paper. https://www.intel.com/content/www/us/en/solid-state-drives/intel-ssd-software-defined-memory-with-vm.html
[8] 2018. The General Atomic and Molecular Electronic Structure System (GAMESS). http://www.msg.ameslab.gov/gamess/index.html
[9] 2018. The Intel Quantum Simulator. https://github.com/intel/Intel-QS
[10] 2018. TrendForce. http://www.trendforce.com
[11] Vitaly A. Vshivkov, Galina G. Lazareva, Alexei V. Snytnikov, I Kulikov, and Alexander V. Tutukov. 2011. Computational methods for ill-posed problems of gravitational gasodynamics. 19 (05 2011).
[12] Berkin Akin, Franz Franchetti, and James C. Hoe. 2016. FFTs with Near-Optimal Memory Access Through Block Data Layouts: Algorithm, Architecture and Design Automation. *J. Signal Process. Syst.* 85, 1 (Oct. 2016), 67–82. https://doi.org/10.1007/s11265-015-1018-0
[13] D. H. Bailey. 1989. FFTs in External of Hierarchical Memory. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing '89)*. ACM, New York, NY, USA, 234–242. https://doi.org/10.1145/76263.76288
[14] Thomas H Cormen and David M Nicol. 1998. Performing out-of-core FFTs on parallel disk systems. *Parallel Comput.* 24, 1 (1998), 5–20.
[15] S. K. Godunov and I. M. Kulikov. 2014. Computation of discontinuous solutions of fluid dynamics equations with entropy nondecrease guarantee. *Computational Mathematics and Mathematical Physics* 54, 6 (01 Jun 2014), 1012–1024. https://doi.org/10.1134/S0965542514060086
[16] I.M. Kulikov, I.G. Chernykh, A.V. Snytnikov, B.M. Glinskiy, and A.V. Tutukov. 2015. AstroPhi: A code for complex simulation of the dynamics of astrophysical objects using hybrid supercomputers. *Computer Physics Communications* 186, Supplement C (2015), 71 – 80. https://doi.org/10.1016/j.cpc.2014.09.004
[17] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.
[18] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (1995), 1–19. https://doi.org/10.1006/jcph.1995.1039
[19] M. V. Popov and S. D. Ustyugov. 2007. Piecewise parabolic method on local stencil for gasdynamic simulations. *Computational Mathematics and Mathematical Physics* 47, 12 (01 Dec 2007), 1970–1989. https://doi.org/10.1134/S0965542507120081
[20] M. V. Popov and S. D. Ustyugov. 2008. Piecewise parabolic method on a local stencil for ideal magnetohydrodynamics. *Computational Mathematics and Mathematical Physics* 48, 3 (01 Mar 2008), 477–499. https://doi.org/10.1134/S0965542508030111
[21] Navin Shenoy. 2018. What Happens When Your PC Meets Intel Optane Memory? https://newsroom.intel.com/editorials/what-happens-pc-meets-intel-optane-memory/
[22] M. Smelyanskiy, N. P. D. Sawaya, and A. Aspuru-Guzik. 2016. qHiPSTER: The Quantum High Performance Software Testing Environment. *ArXiv e-prints* (Jan. 2016). arXiv:quant-ph/1601.07195 https://arxiv.org/abs/1601.07195v2

xBGAS: Toward a RISC-V ISA Extension for Global, Scalable Shared Memory

John D. Leidel
Tactical Computing Laboratories
Muenster, Texas
jleidel@tactcomplabs.com

Xi Wang, Frank Conlon, Yong Chen
Texas Tech University
Lubbock, Texas
xi.wang,frank.conlon,yong.chen@ttu.edu

David Donofrio,Farzad Fatollahi-Fard
Lawrence Berkeley National Laboratory
Berkeley, California
ddonofrio,ffard@lbl.gov

Kurt Keville
MIT Lincoln Laboratory
Cambridge, Massachusetts
klk@mit.edu

ABSTRACT

Given the switch from monolithic architectures to integrated systems of commodity components, scalable high performance computing architectures often suffer from unwanted latencies when operations depart an individual device domain. Transferring control and/or data across loosely coupled commodity devices implies a certain degree of cooperating in the form of complex system software. The end result being a total system architecture the operates in an inefficient manner.

This work presents initial research into creating micro architecture extensions to the RISC-V instruction set that provide tightly coupled support for common high performance computing operations. This xBGAS micro architecture extension provides applications the ability to access globally shared memory blocks directly from rudimentary instructions. The end result being a highly efficient micro architecture for scalable shared memory programming environments.

CCS CONCEPTS

• **Computing methodologies** → *Simulation environments*; *Simulation tools*; • **Computer systems organization** → *Multicore architectures*;

KEYWORDS

RISC-V, instruction set architecture, microarchitecture, shared memory

ACM Reference Format:
John D. Leidel, Xi Wang, Frank Conlon, Yong Chen, David Donofrio,Farzad Fatollahi-Fard, and Kurt Keville. 2018. xBGAS: Toward a RISC-V ISA Extension for Global, Scalable Shared Memory. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3286475.3286478

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11.
https://doi.org/10.1145/3286475.3286478

1 INTRODUCTION

Modern high performance computing system architectures are often constructed using general purpose microprocessors, accelerators, memory devices and high performance, low-latency network architectures. However, despite the advances in fabrication and device packaging technologies, each of the discrete subcomponents is largely architected in a vacuum. As a result, the integration of the scalable system architecture is often governed by layers of software infrastructure. This often induces unwanted latencies, complexity and performance degradations in large-scale parallel applications. Further, given the recent reemergence of extended memory interconnection technologies such as GenZ [7], CCIX [1] and OpenCAPI [2], architects in high performance computing and high performance analytics have sought to exploit these interconnection methodologies for extended or partitioned addressing across device and system architectural domains. Several attempts have been made to classify the benefits of such an approach [11] [14] [10], however, we have yet to see a commercial architecture with native extended addressing capabilities in the ISA garner wide adoption.

In parallel to this, core hardware architecture research has reemerged as a popular research topic. In support of this, several research groups have created reusable and extensible ISA and platform frameworks in order to promote core research activities in hardware architecture. Frameworks such as the RISC-V ISA [19, 20] and the OpenPiton project [4] have emerged as leading candidates for both academic and commercial deployments. As a result, projects such as the GoblinCore-64 data intensive architecture [18], the PULPino low-power SoC [17] and the Shakti-T's light weight security extensions [15] have been developed using more generalized instruction set frameworks.

This work presents a RISC-V ISA extension that lies at the convergence of scalable high performance computing and extensible architecture techniques. The Extended Base Global Address Space, or *xBGAS*, RISC-V extension is designed to provide global, scalable memory addressing support for scalable high performance or enterprise computing architectures. xBGAS provides this extended addressing support via an extended register file that permits architects to extend the base RISC-V RV64 addressing model to support a object-based, flat or partitioned addressing across multiple, distinct nodes. The xBGAS model does so in a manner that supports binary compatibility with traditional RV64 compiled binaries without a

requirement to rebuild the entire software stack. Finally, while the xBGAS extension is not based upon the RV128 addressing model, it can be utilized to implement flat 128 bit addressing.

The remainder of this work is organized as follows. Section 2 discusses previous hardware architectures with similar characteristics. Section 3 presents the xBGAS machine organization, effective addressing and instruction set extension. Section 4 introduces the initial runtime library constructs that mimic the traditional OpenSHMEM [6] library functions. We conclude with a current assessment of our work as well as future xBGAS research activities.

2 PREVIOUS WORK

Several previous high performance computing system architectures have provided similar mechanisms for scalable addressing mechanisms. However, they have done so utilizing proprietary instruction sets or processor architectures. One more recent example is the Convey MX-100 system architecture [12, 13]. The MX-100 system was a heterogeneous architecture that combined traditional x86_64 host processors with a large, dense coprocessor board whose core processing capabilities utilized FPGA's. The memory subsystem on the MX-100 coprocessor board was based upon a unique configuration of custom memory controllers and DDR3 DRAMs that included support for near-memory atomic operations and full/empty bit operations (*tagged* memory). The MX-100 memory subsystem also included a set of additional pins that were designed to accept a daughter board that re-routed a portion of the internal memory links to the rear of the device chassis. These links were designed to support connectivity between up to eight coprocessor devices whereby the memory addressing structure was partitioned both physically and logically. While this functionality was designed as a part of the initial platform, it was never widely deployed.

The second candidate example provides hardware support for scalable memory addressing is the Cray T3D [9, 16]. The T3D architecture contained a DTB Annex [3] that permitted local processors to reference all of the machine's memory without the assistance of remote processors. 43 bit virtual addresses were translated into 34 bit addresses with the addition of a index value into the Annex table. Each of the 32 Annex entries subsequently specified a function code and remote processor (PE) for the target operation. Optionally, two bits were also utilized to specify memory-mapped devices. These physical architecture targets could be utilized by traditional load and store instructions (Alpha ISA) as well as three special memory operations: fetch-and-increment, atomic swap and prefetch. However, given the size limitations of the DTB Annex unit, this pure methodology is not inherently applicable to modern, scalable architectures.

3 RISC-V SCALABLE ADDRESSING

3.1 Application Targets

There are several potential enterprise application targets beyond scalable high performance computing whereby the xBGAS RISC-V extension may be applied. While this work specifically focuses on applying the xBGAS extension for high performance computing, we may summarize the additional design goals and operational applications as follows:

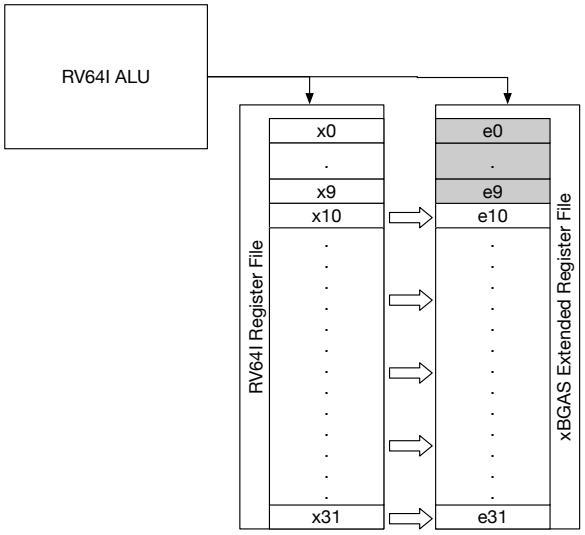


Figure 1: xBGAS Machine Organization

- **HPC-PGAS**: Our specific research focuses on the ability to construct high performance computing instruments (HPC) with partitioned global addressing (PGAS). Programming models such as UPC and Chapel provide users the ability to explicitly denote locality in the construction and distribution of their data members as a part of the language specification. Rather than providing this functionality using a complex runtime library, we seek to utilize xBGAS to provide machine-level support for HPC-PGAS.
- **HPA-FLAT**: High performance analytics (HPA) problems may also utilize xBGAS where graph or non-deterministic data structures cannot be efficiently sharded. In this manner, xBGAS may provide a flat addressing model similar in design to RV128.
- **MMAP-IO**: We may also utilize xBGAS to provide memory mapped filesystem support for a variety of data intensive or file serving needs. Efficient mapping of inode tables into the extended address range may provide significant performance advantages when accessing large-scale file systems.
- **Cloud-BSP**: Cloud computing programming models may also utilize xBGAS. Given the widespread adoption of Java in cloud computing models, xBGAS provides a path by which to provide support for memory-mapped objects and global object visibility without the need to provide full 128-bit addressing in the Java JVM.

3.2 xBGAS Machine Organization

One of the driving design goals for the xBGAS RISC-V extension is the natural ability to execute unmodified R64I binary blobs. In this manner, xBGAS-enabled devices have the ability to boot and execute RISC-V Linux with all the ancillary kernel modules without issue. The key portions of the xBGAS functionality and associated addressing modes that provide this functionality are implemented via an extended register file that is mapped to the base RISC-V register file (Figure 1). These registers are only visible and utilized

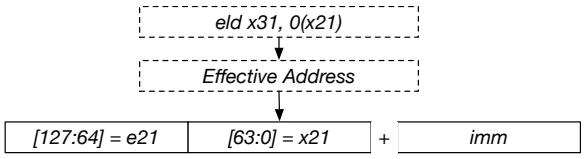


Figure 2: xBGAS Effective Addressing

by the extended xBGAS instructions. In this manner, the RISC-V core is configured to support the standard register width (*XLEN* in the RISC-V vernacular) of 64 bits.

In Figure 1, you'll find that for each base register (*xN*), we map an equivalent extended register denoted as *eN*. The indexing for the extended register file is mapped to the index of its complementary base register. However, also note that the first ten extended indices (*0-9*) cannot be utilized for addressing. These indices are reserved for operations in the memory translation layer. This is analogous to the first ten indices in the general purpose register file that are specifically allocated to various operations associated with instruction handling (*sp*), frame handling (*fp*), context save/restore and hard encoding (*zero/x0*) as defined by the standard RV64 ABI. As a result, these registers cannot be utilized in generating extended addresses for xBGAS memory operations. Any attempt to utilize extended registers below index 10 will result in an exception.

3.3 xBGAS Addressing

Given the aforementioned machine organization, we define the xBGAS addressing methodology such that the base register file contains the address of the target data block in the desired addressing mode and the extended register contains an object ID that denotes the logical storage location for that address. The logical object ID's can be mapped to physical node identities, physical address spaces, storage devices (block devices), memory mapped file systems or other storage mediums. We also note that the extended registers can be utilized for flat, 128 bit address spaces. However, we do not currently have an appropriate application where this is deemed necessary.

Under normal operations where the general purpose register file is utilized for addressing, the extended registers are ignored and the default RV64 addressing mode is utilized. However, when the processor encounters an xBGAS extended instruction, the extended register at the target index is utilized for the effective address calculation. In Figure 2, we encounter an extended load operation whose base address is found in the *x21* register offset by the immediate value *0*. The result is stored in *x31*. However, when the effective address is generated, the base register is utilized for the target address and the *e21* extended register at the same index is utilized for the object ID. In this manner, we maintain virtual addressing across the system and permit the remote (target) node to perform any virtual to physical translation, page manipulation and enforce any necessary access control.

3.4 xBGAS ISA Extension

Utilizing the xBGAS machine organization and effective addressing structure, we provide three sets of extended instructions. Note that these instructions require that the base RV64I instruction set and

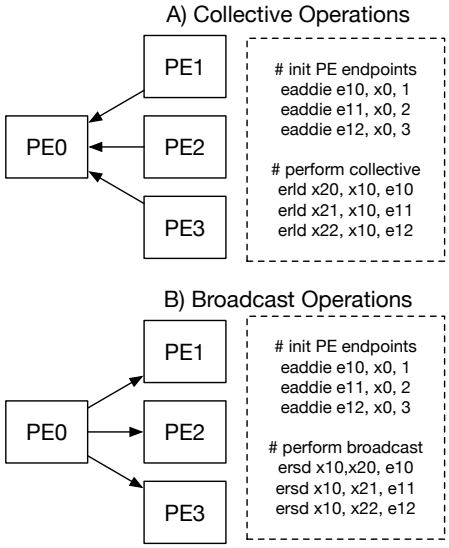


Figure 3: xBGAS Collectives and Broadcasts

associated functionality is present and functional per the official RISC-V instruction set specification [19]. We can summarize these extensions as follows:

- **Address Management**: The address management instructions include three basic instructions that provide users the ability to directly manipulate the values within the extended registers. These instructions follow the standard RISC-V *I-type* instruction encoding and utilize the core RISC-V ALU much in the same manner as moving data between general purpose registers. With these instructions, users have the ability to utilize the extended registers as the source, target or source and target for unsigned integer arithmetic. This implies that moving values between registers is done using basic integer arithmetic similar to the following:

```
eaddi x10, e22, 0  # Set x10 = e22 + 0
eaddie e22, x10, 0 # Set e22 = x10 + 0
eaddix e22, e21, 0 # Set e22 = e21 + 0
```

- **Integer Load/Store Instructions**: Much in the same manner as the base RV64I instruction set, we provide signed and unsigned loads and stores for all the base integer types up to 64 bits in width. These instructions are organized in the standard RISC-V *I-type* encoding format such that immediate values can be utilized as offsets to the base (general purpose register) address.
- **Raw Integer Load/Store Instructions**: The final type of instruction contained within the xBGAS specification is the raw integer load and store instructions. Unlike the previous types, the raw integer load and store instructions utilize the RISC-V *R-type* encoding. These instructions permit special cases of extended loads and stores whereby the extended register utilized for extended addressing can be explicitly specified. This permits applications to perform more complex operations within the higher level programming model. As we see if Figure 3, these instructions can be utilized to

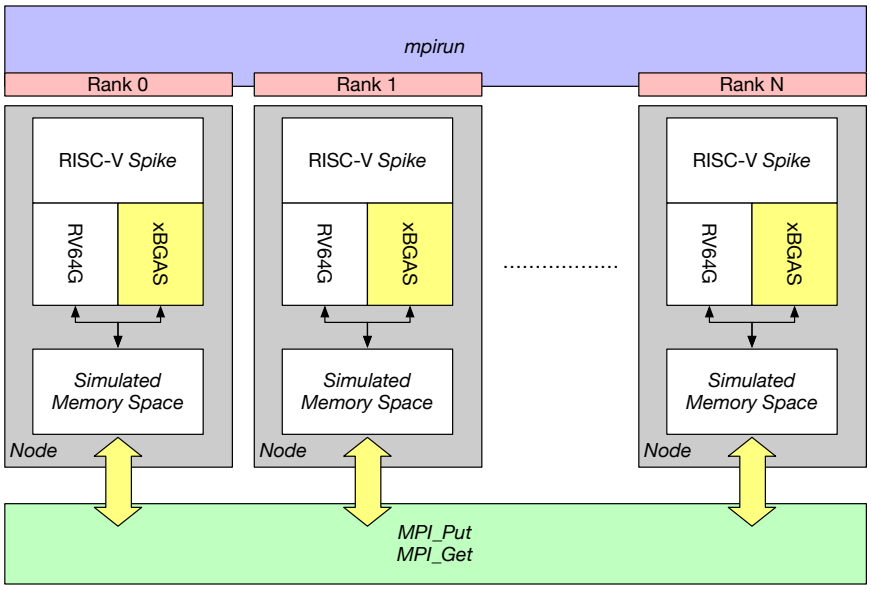


Figure 4: xBGAS Simulation Environment

create collective and broadcast constructs by simply manipulating the values in the individual extended registers. The first example (Figure 3.A) depicts an example of a collective operation. *PE0* initiates the operation by first initializing the PE endpoints using `eaddie` instructions. Next, *PE0* performs the collective operation by issuing three extended register load operations (`erld`) using the three initialized PE's. The second example (Figure 3.B) depicts an example of a broadcast operation. Similar to our collective operation, *PE0* initializes the endpoints in registers `e10-e12`. The broadcast is actually performed when *PE0* issues the extended register store operations via the `ersd` instructions. Note how PE0 utilizes the same value residing in register `x10` to store to each of the remote PE's.

4 RUNTIME INFRASTRUCTURE

4.1 Simulation Environment

The xBGAS compiler, tools and functional simulator are based upon the mainline RISC-V tools. The simulator is based upon the traditional RISC-V functional simulator, *Spike*. We have augmented the simulator in two major areas. First, we have added support for all of the aforementioned xBGAS and register extensions within the core simulator. We have further verified that traditional RV64 binary payloads continue to function as expected with these modifications.

Second, we have further augmented the xBGAS Spike simulator to provide scalable simulation capabilities (Figure 4). The core simulator has been augmented to support MPI messaging within the memory simulation functions such that xBGAS instructions have the ability to initiate shared memory requests between multiple instantiations of xBGAS Spike simulator. In this manner, the xBGAS integrated simulation infrastructure is only limited by the scalability of the parallel infrastructure on which it is deployed.

4.2 xBGAS Runtime

The xBGAS runtime infrastructure is designed to provide machine-level functionality via a C-based library such that users and applications may perform memory allocation and data motion in the form of *gets* and *puts* via the xBGAS instruction set extensions. The library is architected to serve as a machine-level runtime layer under more traditional programming models such as OpenSHMEM [6], UPC++ [8] and Chapel [5].

The xBGAS runtime infrastructure is implemented via a combination of C and assembly language. The assembly language functions are utilized to implementation high performance utility functions such as querying the environment as well as high performance data motion functions. The runtime infrastructure provides four main functions summarized as follows:

- **Constructor**: The constructor routines perform the environment initialization prior to the application encountering the *main* function. In this manner, all of the PE endpoints have been initialized and the logical to physical mapping tables for the xBGAS object references have been initialized.
- **Memory Allocation**: The memory allocation routines are designed to mimic the OpenSHMEM *shmalloc* notion of allocating congruent blocks of memory on each participating PE. Currently, we require minimum allocations of at least a 4KiB page in order to avoid unnecessary paging when accessing remote memory.
- **Environment**: The environment routines provide basic information back to the calling application. This includes the logical PE identifier, the number of participating PEs and the ability to query whether an address is accessible on a remote PE.
- **Data Motion**: The final block of routines provides rudimentary data motion in the form of *gets* and *puts*. The data motion routines support all the rudimentary types defined

in the OpenSHMEM specification [6]. The routines are also provided with blocking and non-blocking versions. Given the native weak memory ordering of many of the RISC-V implementations, we have made every effort to produce high performance implementations of the data motion routines that are manually unrolled. As a result, any time a data motion is requested for at least eight elements (regardless of the size of the element), the runtime library will utilize a manually unrolled data motion routine written in assembly to perform the actual transfer.

5 CONCLUSIONS

In conclusion, this work presents the initial research behind a scalable, globally shared memory micro architecture extension to the RISC-V instruction set. The xBGAS infrastructure provides basic extensions to the core register infrastructure and instruction set definitions whereby applications have the ability to directly access remote memory blocks via rudimentary instructions. Further, the xBGAS infrastructure provides this support in a manner that maintains the ability to execute pure RV64I binaries without modification. As a result, our extended RISC-V infrastructure has the immediate ability to boot and execute a full Linux operating system.

In addition to the basic hardware specification, we also provide an initial simulation environment and associated runtime infrastructure such that we have the ability to begin porting more expressive programming models such as OpenSHMEM, UPC++ and Chapel. These software utilities will assist in initial performance characterizations and hardware prototyping efforts.

6 FUTURE WORK

While this work does not contain specific results regarding the performance of the xBGAS infrastructure, we have demonstrated functional tests using the extended instructions on the xBGAS simulator. In addition, we have also demonstrated a functional Linux kernel executing on the simulator. Future work will be done to demonstrate the performance ramifications of the extended instructions on various different networks. This performance prototyping will likely be done in conjunction with the Sandia Structural Simulation Toolkit (SST) *Stake* simulation infrastructure that provides RISC-V simulation capabilities in SST. In this manner, we have the ability to model cycle-based compute and network traffic using various latency configurations, topologies and backing memory stores.

In addition to the aforementioned performance modeling, we also seek to develop a hardware implementation of the xBGAS extensions. The hardware prototype is currently being designed in conjunction with the RISC-V Rocket implementation in Chisel HDL. The initial deployment target for the xBGAS infrastructure will target a high performance FPGA platform.

ACKNOWLEDGMENTS

The authors would like to thank the following individuals for their assistance in reviewing various portions of the xBGAS specification. Dr. Bruce Jacobs (University of Maryland) for his help in reviewing the feasibility of virtual to physical memory translation, Mr. John Shalf (Lawrence Berkeley National Laboratory) for his help in reviewing application scenarios and Mr. Steven Wallach (Micron) for his help in reviewing the application to PGAS programming environments.

REFERENCES

[1] [n. d.]. CCIX Consortium. https://www.ccixconsortium.com/. Accessed: 2017-09-09.
[2] [n. d.]. OpenCAPI Consortium. http://opencapi.org/. Accessed: 2017-09-09.
[3] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. 1995. Empirical evaluation of the CRAY-T3D: a compiler perspective. In *Proceedings 22nd Annual International Symposium on Computer Architecture*. 320–331.
[4] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 217–232. https://doi.org/10.1145/2872362.2872414
[5] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *IJHPCA* 21 (2007), 291–312.
[6] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10)*. ACM, New York, NY, USA, Article 2, 3 pages. https://doi.org/10.1145/2020373.2020375
[7] GenZ Consortium. 2017. *GenZ Core Specification*. Technical Report. GenZ Consortium. http://genzconsortium.org/specifications/draft-core-specification-july-2017/
[8] UPC++ Specification Working Group. 2018. *UPC++ Specification v1.0 Draft 5*. Technical Report. Lawrence Berkeley National Laboratory.
[9] Vijay Karamcheti and Andrew A. Chien. 1995. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. *SIGARCH Comput. Archit. News* 23, 2 (May 1995), 298–307. https://doi.org/10.1145/225830.224440
[10] John D Leidel, , Xi Wang, and Yong Chen. 2017. Toward a Memory-Centric, Stacked Architecture for Extreme-Scale, Data-Intensive Computing. In *Workshop On Pioneering Processor Paradigms, 2017 IEEE Symposium on High Performance Computer Architecture*. IEEE.
[11] John D. Leidel. 2017. *GoblinCore-64: A Scalable, Open Architecture for Data Intensive High Performance Computing*. Ph.D. Dissertation. Texas Tech University.
[12] J. D. Leidel, J. Bolding, and G. Rogers. 2013. Toward a Scalable Heterogeneous Runtime System for the Convey MX Architecture. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. 1597–1606. https://doi.org/10.1109/IPDPSW.2013.18
[13] J. D. Leidel, K. Wadleigh, J. Bolding, T. Brewer, and D. Walker. 2012. CHOMP: A Framework and Instruction Set for Latency Tolerant, Massively Multithreaded Processors. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 232–239. https://doi.org/10.1109/SC.Companion.2012.39
[14] John D. Leidel, Xi Wang, and Yong Chen. 2015. *GoblinCore-64: Architectural Specification*. Technical Report. Texas Tech University. http://gc64.org/wp-content/uploads/2015/09/gc64-arch-spec.pdf
[15] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. 2017. Shakti-T: A RISC-V Processor with Light Weight Security Extensions. In *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP '17)*. ACM, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/3092627.3092629
[16] Wilfried Oed and Martin Walker. 1993. An Overview of Cray Research Computers Including the Y-MP/C90 and the New MPP T3D. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '93)*. ACM, New York, NY, USA, 271–272. https://doi.org/10.1145/165231.165266
[17] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K. Gurkaynak, and Luca Benini. 2016. PULPino: A small single-core RISC-V SoC. http://iis-projects.ee.ethz.ch/images/d/d0/Pulpino_poster_riscv2015.pdf RISC-V Workshop.
[18] Xi Wang, John D. Leidel, and Yong Chen. 2016. Concurrent Dynamic Memory Coalescing on GoblinCore-64 Architecture. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. ACM, New York, NY, USA, 177–187. https://doi.org/10.1145/2989081.2989128
[19] Andrew Waterman and Krste Asanovic. 2017. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. Technical Report. SiFive, Inc. https://riscv.org/specifications/
[20] Andrew Waterman and Krste Asanovic. 2017. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. Technical Report. SiFive, Inc. https://riscv.org/specifications/

Understanding Application Recomputability without Crash Consistency in Non-Volatile Memory

Jie Ren
University of California, Merced
jren6@ucmeced.edu

Kai Wu
University of California, Merced
kwu42@ucmerced.edu

Dong Li
University of California, Merced
dli35@ucmerced.edu

ABSTRACT

Emerging non-volatile memory (NVM) is promising to be used as main memory, because of its good performance, density, and energy efficiency. Leveraging the non-volatility of NVM as main memory, we can recover data objects and resume application computation (recomputation) after the application crashes. The existing work studies how to ensure that data objects stored in NVM can be recovered to a consistent version during system recovery, a property referred to as crash consistency. However, enabling crash consistency often requires program modification and brings large runtime overhead.

In this paper, we use a different view to examine application recomputation in NVM. Without taking care of consistency of data objects, we aim to understand if the application can be recomputable, given possible inconsistent data objects in NVM. We introduce a PIN-based simulation tool, NVC, to study application recomputability in NVM without crash consistency. The tool allows the user to randomly trigger application crash and then perform postmortem analysis on data values in caches and memory to examine data consistency. We use NVC to study a set of applications. We reveal that some applications are inherently tolerant to the data inconsistency problem. We perform a detailed analysis of application recomputability without crash consistency in NVM.

ACM Reference Format:
Jie Ren, Kai Wu, and Dong Li. 2018. Understanding Application Recomputability without Crash Consistency in Non-Volatile Memory. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3286475.3286476

1 INTRODUCTION

Emerging byte-addressable non-volatile memory (NVM) technologies, such as memristors [21] and spin-transfer torque MRAM (STT-MRAM) [12], provide better density and energy efficiency than DRAM. Those memory technologies also have the durability of the hard drive and DRAM-like performance. Those properties of NVM allow us to use NVM as main memory, which blurs the traditional divide between byte-addressable, volatile main memory and block-addressable, persistent storage.

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286476

Leveraging the non-volatility of NVM as main memory, we can recover data objects and resume application computation (recomputation) after the application crashes. However, with write-back caching, stores may reach NVM out of order, breaking data object recoverability. Also, data objects cached in the cache hierarchy and stored in NVM may not be consistent. Such inconsistence persists after the application restarts and may impact application execution correctness. Consequently, many existing work [2, 3, 7, 15, 17, 28] studies how to ensure that data objects stored in NVM can be recovered to a consistent version during system recovery, a property referred to as *crash consistency*.

To ensure crash consistency, the programmer typically invokes ISA-specific cache flushing mechanisms via inline assembly or library calls (e.g., `CLFLUSH`) to ensure persist order. To enable crash consistency, log-based mechanisms and the checkpoint mechanism [3, 6, 23, 29] are often employed to make a copy of critical data objects. However, frequent cache flushing, data logging and checkpointing can cause program stalls and large runtime overhead [33].

In this paper, we use a different view to examine application recomputation in NVM. Without taking care of consistency of data objects, we aim to understand if the application can be recomputable, given possible inconsistent data objects in NVM. We define application recomputability regarding *application outcome correctness*. In particular, we claim an application is recomputable after a crash if the application outcome is correct. If an application is recomputable without crash consistency of data objects, then we do not need to employ any cache flushing or logging mechanisms, which improves performance. Having the performance improvement is especially attractive to applications in the field of high-performance computing (HPC).

Evaluating application recomputability without crash consistency is not trivial, because of the following reasons. *First*, to evaluate application recomputation, we must collect data object values in NVM for recomputation, when the crash happens. Without available NVM hardware, the traditional DRAM-based main memory, although often used to emulate NVM [27], can lose data when the crash happens. NVDIMM-N provide a possible solution to address the problem. In particular, when a power failure happens, NVDIMM-N copies the data from the volatile traditional DRAM to flash storage and copies it back when power is restored. In the solution of NVDIMM-N [24], the traditional DRAM is used to emulate NVM, and a small backup power source (e.g., a large battery) is used to make the data copy during the power failure. However, NVDIMM-N is not suitable for our evaluation, because our evaluation involves a large amount of application crash tests. For those tests, a machine with NVDIMM-N has to repeatedly stop and restart, which is time-consuming and impacts the machine reliability.

Second, we must determine data consistency when the crash happens. This requires that we compare data in caches and its counterpart in memory. This indicates that we must track data dirtiness of each cache line in caches. To quantify how much inconsistency there is between two data copies, we must also record data values of dirty cache lines. The real hardware does not allow us to track data values and dirtiness of cache lines. The existing simulators usually do not store data values in caches and main memory for simulation.

To evaluate application recomputability without crash consistency, we introduce a PIN [19]-based simulation tool, named *NVC* (standing for *N*on-*V*olatile memory *C*rash tester). In essence, the tool is a PIN-based cache simulator plus rich functionality for crash tests. The tool allows the user to randomly trigger application crash and then perform postmortem analysis on data values in caches and memory to examine data consistency. The tool associates rich data semantics with data values, such that the user can determine which data objects are critical to application recomputatbility. The tool is highly customizable, allowing the user to configure cache hierarchy, cache coherence, and the association between data values and data semantics. The tool also allows the user to test the impact of different cache flushing mechanisms (e.g., `CLFLUSH`, `CLWB` and `CFLUSHOPT`) on data consistency. The tool also integrates the functionality of restarting the application with postmortem data in memory to determine application recomputability.

NVC is useful for several scenarios. Beyond being used to study application recomputability, NVC can be used as a debugger tool. As a debugger tool, NVC can be used to examine if the persist order enforced by the programmer is correct. It can also be used to detect if the data value of a specific variable is consistent as expected by the programmer when the application crashes.

We use NVC to study recomputability of several representative applications from the fields of HPC and machine learning. Using thousands of crash tests, we statistically reveal that some applications do not need crash consistency on critical data objects and are highly recomputable after crashes. We study the reasons that account for the application's inherent tolerance to crash consistency, including memory access pattern, data size, and application algorithm.

The major contributions of the paper are summarized as follows.

- We introduce a tool, NVC, to study application recomputability in NVM without crash consistency, which is unprecedented.
- We use NVC to study a set of applications. Different from the existing work that relies on enforcing crash consistency for application recomputation, we reveal that some applications are inherently tolerant to crash consistency. We perform a detailed analysis of the reasons.

2 PROBLEM DEFINITION AND BACKGROUND

Definition of data objects. Data objects considered in the paper refer to large data structures that store computation results. An example of such data structures is multi-dimensional arrays that represent large matrices. When running an application in a large-scale parallel system, those data objects are often the target to apply checkpoint. In a programming model for NVM (e.g., PMDK [14]), those data objects can be placed into a memory (NVM)-mapped file for the convenience of application restart.

In this paper, we do not consider memory address-related crash consistency. The memory address-related crash consistency problem can cause dangling pointers, multiple frees, and memory leaks. Those problems can easily cause memory segmentation fault when the application restarts. Many existing efforts (e.g., [6, 29]) can address the memory address-related crash consistency. Those efforts are complementary to our work.

In addition, we do not consider those applications with strong demand for *memory transactions*. Those applications include transactional key-value store and the relational database. For those applications, losing data consistency has a severe impact on the functionality of those applications, although some of them usually do not have any problem to restart after crashes.

Application recomputability. We define application recomputability in terms of application outcome correctness. In particular, we claim an application is recomputable after a crash, if the application can restart and the final application outcome remains correct.

The application outcome is deemed correct, as long as it is acceptable according to application semantics. Depending on application semantics, the outcome correctness can refer to precise numerical integrity (e.g., the outcome of a multiplication operation must be numerically precise), or refer to satisfying a minimum fidelity threshold (e.g., the outcome of an iterative solver must meet certain convergence thresholds).

We distinguish restart and recomputability in the paper. After the application crashes, the application may resume execution, which we call *restart*, but there is no guarantee that the application outcome after the application restarts is correct. *If the application outcome is correct, we claim application is recomputable.*

Application restart. When the application crashes, data objects that are placed into a memory (NVM)-mapped file are persistent and usable to restart the application. Other data objects in NVM, either being consistent or inconsistent, are not used for application restarting.

Typically it is the programmer's responsibility to decide which data objects should be placed into the file. Those data objects are critical to application execution correctness. We name those data objects as *critical data objects* in the paper. In many applications, non-critical data objects are either read-only or can be recomputed based on the critical data objects.

In our study, we focus on applications with iterative structures. In those applications, there is a main computation loop dominating computation time. We choose those applications because they are promising to be recomputable after crashes: The iterative structures of those applications may allow the computation of those applications to amortize the impact of corrupted critical data objects. There are a large amount of those applications, including most HPC applications and many machine learning training algorithms.

3 NVC: A TOOL FOR STUDYING APPLICATION RECOMPUTABILITY

NVC is a PIN-based crash simulator. NVC simulates a multi-level cache hierarchy with cache coherence and main memory; NVC also includes a random crash generator, a set of APIs to support the configuration of crash tests and application restart, and a component to examine data inconsistency for post-crash analysis. For the simulation of cache and main memory, different from the traditional PIN-based cache simulator, NVC not only captures microarchitecture level, cache-related hardware events (e.g., cache misses and cache invalidation), but also records the most recent value of data objects in the simulated caches and main memory.

NVC is highly configurable and supports a range of crash tests with different configurations, summarized as follows.

- **Cache configuration**, including the selection of a cache coherence protocol and typical microarchitecture configurations (e.g., cache associativity and cache size);
- **Crash configuration**, including when to trigger the crash and which data objects are critical;
- **Cache flush configuration**, including specifying which cache flushing instruction will be used to ensure data consistency;
- **Recomputation configuration**, including specifying a point within a program for restarting.

We describe the main functionality of NVC as follows.

Cache simulation. Besides supporting the simulation of multi-level, private/shared caches with different capacities and associativity, our cache simulation supports the simulation of cache coherence, which allows us to study the impact of cache coherence on data consistency. With the deployment of a cache coherence protocol, it is possible that a private cache has a stale copy of a cache block, while NVM has the most updated one, causing data inconsistency. NVC can capture such data inconsistency and ignore it if configured to do so. In our evaluation, we use data in NVM to restart and does not count such data inconsistency, because NVM has the most updated data values.

Using PIN to intercept every memory read and write instructions from the application, NVC can get memory addresses and corresponding data values associated with memory accesses. NVC also records cache line information, such as data values in each cache line, cache line dirtiness, and validness.

Our cache simulation supports different cache flushing mechanisms. In particular, we provide three APIs: *flush_cache_line()*, *cache_line_write_back()* and *write_back_invalidate_cache()*. Table 1 contains more details.

Main memory simulation. Different from the traditional microarchitectural simulation for main memory, the main memory simulation in NVC aims to record data values. In particular, NVC uses a hashmap with memory addresses as keys and data values as values. Using the hashmap enables easy updates of data values: whenever the cache simulation in NVC writes back any cache line, the main memory simulation can easily find the corresponding record in the simulated main memory.

Random crash generation. NVC emulates the occurrence of a crash by stopping application execution after a randomly selected instruction. To allow the user to limit crash occurrence to a specific code region (e.g., a function call or a loop), NVC introduces two functions, *start_crash()* and *end_crash()*, to delineate the code region. NVC intercepts the invocations of the two functions to determine where to trigger a crash. To statistically quantify application recomputability, we perform a large number of crash tests (thousands of tests) per benchmark.

To enforce random crash generation, NVC profiles the total number of instructions (specified as N), before crash tests. For each crash test, NVC generates a random number n ($1 \leq n \leq N$). After n instructions are executed, NVC stops application execution. Furthermore, NVC has functionality to report call path information when a crash happens. This is implemented by integrating CCTLib [1] into NVC. CCTlib is a PIN-based library that collects calling contexts during application execution. The call path information is useful for the user to interpret crash results. In particular, the call path information introduces the program context information for analyzing crash results. Having the context information is useful to distinguish those crash tests that happen in the same program position (i.e., the same program statement), but with different call stacks.

Data inconsistent rate calculation. NVC reports data inconsistent rate after a crash happens. The data inconsistent rate is defined in terms of either all data in main memory or specific data objects. If the data inconsistent rate is for all data in main memory, then the data inconsistent rate is the ratio of the number of inconsistent data bytes to the size of whole memory footprint of the application. If the data inconsistent rate is for specific data objects, then the data inconsistent rate is the ratio of the number of inconsistent data bytes of the specific data objects to the size of the data objects.

We use the following method to calculate the data inconsistent rate. We distinguish cache line and cache block in the following discussion. The cache line is a location in the cache, and the cache block refers to the data that goes into a cache line. When a crash happens, NVC examines cache line status in the simulated cache hierarchy. If a cache line in a private cache has "invalidate" status, this cache line is not considered for the calculation of data inconsistence rate, because either another private cache or main memory has an updated version of the cache line data and inconsistent data rate will be based on the new version of the cache line data. If a cache line has "dirty" status, then NVM compares the dirty cache block of the cache line with the corresponding data in main memory to determine the number of dirty data bytes. Note that for a specific dirty cache block, we only consider it once, even if the cache block may correspond to multiple cache lines in the cache hierarchy.

To calculate the data inconsistent rate for the critical data objects, NVC must know memory addresses and data types of those data objects, such that we can determine if a cache line has data of the critical data objects. NVC relies on the user to use a dummy function, *critical_data()*, to pass memory address and data type information of a data object to NVC. This function is nothing but uses memory address and data type as function arguments. NVC intercepts them and associate them with a critical data object.

Application restart. When restarting the application, NVC reads the critical data objects, initializes other data objects using the initialization function of the application, and then resumes the

Table 1: APIs for using NVC.

Signature	Description
void start_crash(); void end_crash();	Define where a crash could happen. A crash could happen within the code region encapsulated by the two APIs.
critical_data(void const *p, char type[], int const size);	Collect the address, type and size information of a critical data object.
consistent_data(void const *p, char type[], int const size);	Collect the address, type and size information of a consistent data object.
void cache_line_write_back(void const *p);	Writes back a dirty cache line containing the address p, and marks the cache line as clean in the cache hierarchy. This API is used to emulate CLWB.
void flush_cache_line(void const *p);	Flush a cache line containing address p, invalidate this cache line from every level of the cache hierarchy in the cache coherence domain. This API is used to emulate CLFLUSH and CLFLUSHOPT.
void write_back_invalidate_cache()	Writes back all dirty cache lines in the processor's cache to main memory and invalidates (flushes) the cache hierarchy. This API is used to emulate WBINVD.

main computation loop. Note that when NVC restarts the application, except the critical data objects, other data objects are not usable, even though NVC has data values for those data objects. This is because data semantics for those data values are lost. NVC does not know which data values belong to which data objects.

In our crash tests, we restart the application from the iteration of the main loop where the crash happens, instead of recomputing the whole main loop. To know which iteration to restart, we flush the cache line that contains the iterator at the end of each iteration to make the iterator consistent.

Putting all together. Figure 1 generally depicts the workflow of using NVC. To use NVC, the user needs to insert specific APIs to specify critical data objects, the initialization phase of the application for a restart, and specific code regions for crash tests. The user also needs to configure cache simulation and crash tests. During the application execution, NVC leverages the infrastructure of PIN to instrument the application and analyze instructions for cache simulation. NVC triggers a crash as configured and then perform post-crash analysis to report data inconsistent rate and then restart the application.

An example case. We take MG as an example to explain how we perform a crash test. We use the same method to preform crash tests for other benchmarks.

Figure 2 shows how we add NVC APIs into MG. MG has two critical data objects and their information is passed to NVC in Line 7 and Line 8. The crash test happens in the main computation loop encapsulated by *start_crash()* and *end_crash()*. Right before the main computation loop, we flush whole cache hierarchy (Line 11) to ensure that all data is consistent before we start the crash test. Within the main loop, we flush the cache line containing the iterator at the end of each iteration (Line 15) for the convenience of application restart.

```
1  ...
2  static double u[NR];
3  static double r[NR];
4  ...
5  int main(int argc, char **argv) {
6    int it;
7    critical_data(&u[0], "double", NR);
8    critical_data(&r[0], "double", NR);
9    consistent_data(&it, "int", 1);
10   ...
11   write_back_invalidate_cache();
12   start_crash();
13   for (it = 1; it <= nit; it++) {
14         ...
15     cache_line_write_back(&it);
16   }
17   end_crash();
18   ...
19 }
```

Figure 2: Add NVC APIs into MG

4 RECOMPUTABILITY EVALUATION

4.1 Execution Platform and Simulation Configurations.

In this paper, we simulate a two-level, inclusive cache hierarchy, using the LRU replacement policy. The first level is a private cache (256KB per core and we simulate 8 cores). The second level is a shared cache (20MB). In addition, we use a write-back and no-write allocate policy for the first level cache, and a write-back and write allocate policy for the second level cache. The cache line size is 64 bytes for both caches.

4.2 Benchmark Background.

We use three benchmarks from NAS parallel benchmark (NPB) suite 3.3.1 and one machine learning code (Kmeans [4]) for our study. Those benchmarks are summarized in Table 2. For each benchmark, we use two different input problems, such that we can study the impact of different memory footprint sizes on application recomputability. We describe the benchmarks in details in this section.

Conjugate gradient method (CG). CG is used to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive definite matrix. CG is an iterative method, in the sense that it starts with an imprecise solution and then iteratively converges towards a better solution. CG has a verification phase at the end of CG. The verification tracks the solution convergence towards

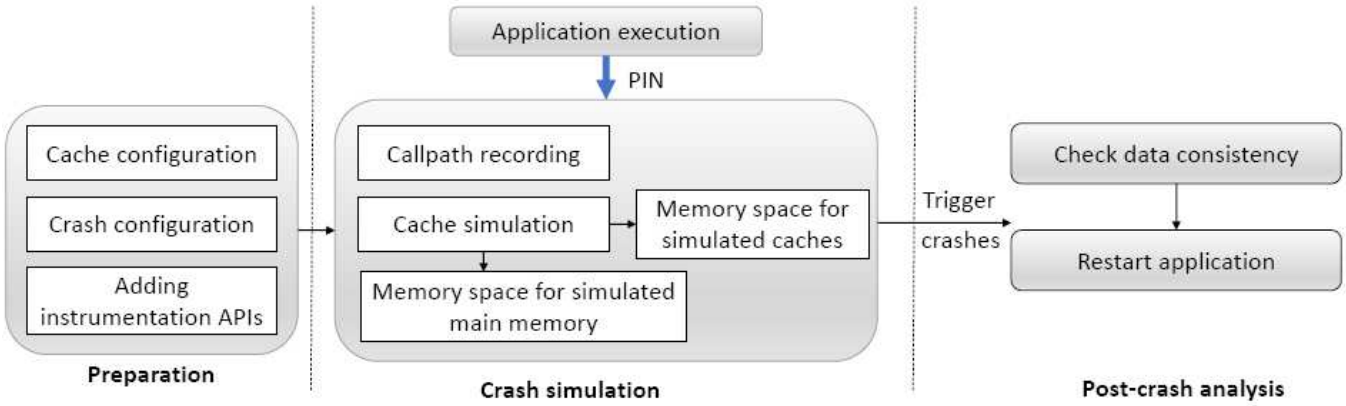
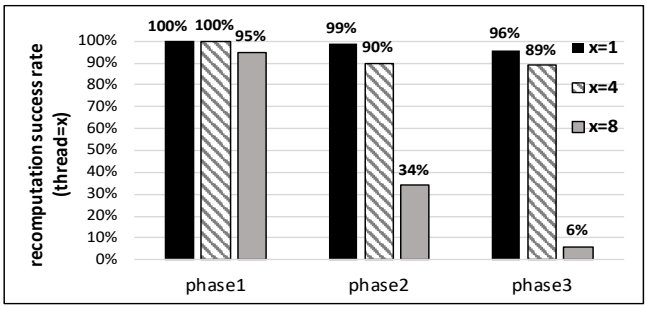
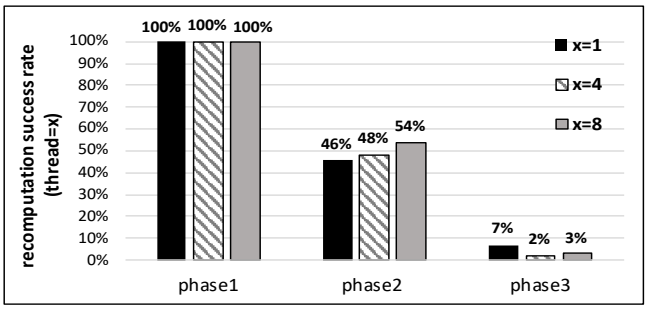


Figure 1: The general workflow of using NVC.

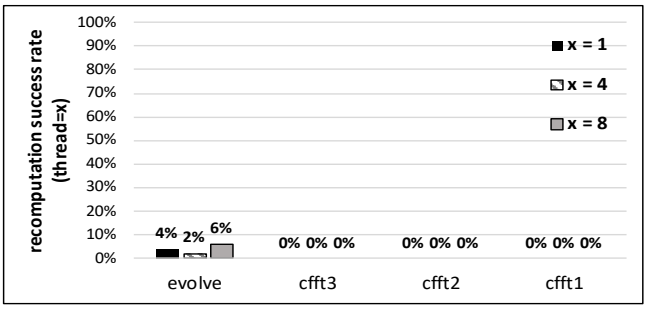


(a) CLASS=A

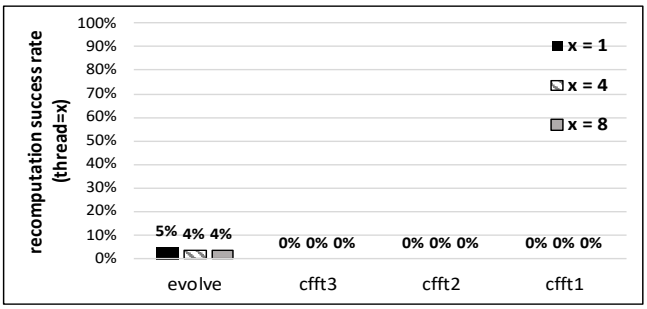


(b) CLASS=B

Figure 3: Recomputation success rate for CG.



(a) CLASS=A



(b) CLASS=B

Figure 4: Recomputation success rate for FT.

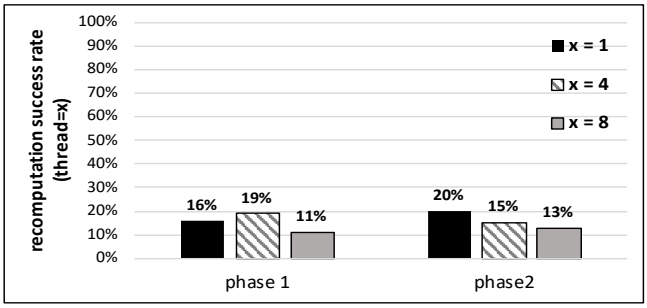
a precision solution. We determine if CG successfully recomputes based on the CG verification.

CG has sparse, unstructured matrix vector multiplication with irregular memory access patterns. In CG, we have five critical data objects (x, p, q, r and z), taking 1% of total memory footprint size.

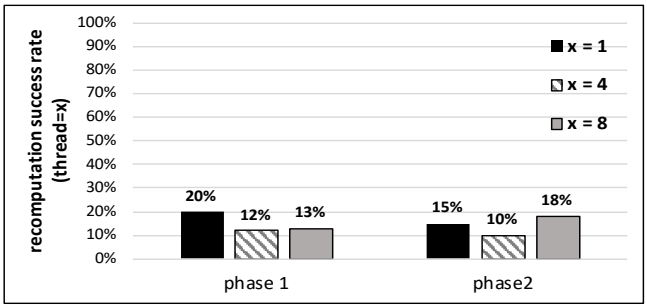
Fourier transform (FT). This benchmark solves a partial differential equation using forward and inverse fast Fourier transform (FFT). FT has a main loop repeatedly performing FFT. At the end of each iteration of the main loop, FT has a verification phase to examine the result correctness of each iteration. The verification phase compares some checksums embedded in data objects of FT with reference checksums to determine the result correctness. Since FT is not an iterative solver and has verification at each iteration of the main loop, we simulation one iteration for our study, in order to save simulation time. We determine if FT successfully recomputes based on the FT verification.

FT has strided memory access patterns. Depending on the input problem size of FT, the stride size can be large, causing intensive accesses to main memory. In FT, we have two critical data objects ($u0$ and $u1$), taking at least 80% of total memory footprint size.

Multigrid method (MG). MG is used to obtain an approximation solution to the discrete Poisson problem based on the multigrid method. MG is also an iterative method, in the sense that MG alternatively works at finer or coarser variants of the input problem

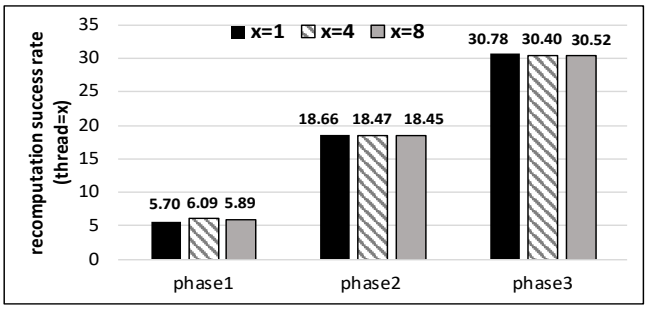


(a) CLASS=A

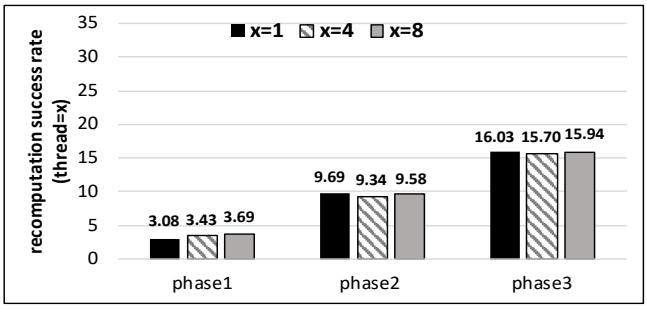


(b) CLASS=B

Figure 5: Recomputation success rate for MG.



(a) kdd as input



(b) 8192 as input

Figure 6: Recomputation success rate for Kmeans.

to compute the increasingly more accurate solutions. MG has a verification phase at the end of MG. The verification tracks the solution convergence towards a precision solution. We determine if MG successfully recomputes based on the MG verification.

MG has either short or long distance data movement, depending on whether MG works on finer or coarser granularity of the input problem. Working at the coarse granularity, MG can cause intensive main memory accesses. In MG, we have two critical data objects (u and r), taking a least 70% of total memory footprint size.

Kmeans. Kmeans is a clustering algorithm to classify a set of input data points into n clusters. Kmeans is an iterative algorithm: In each iteration, each data point is associated with its nearest cluster, and then a new cluster centroid for each cluster is formed. Kmeans iteratively determines the cluster centroid for each cluster. Kmeans will stop clustering when a convergence criteria is met. Given a set of data points as input, the algorithm identifies related points by associating each data point with its nearest cluster, computing new cluster centroids and iterating until converge.

Kmeans has streaming memory accesses (lacking temporal data locality). In Kmeans, we have *centers* as critical data objects, which is the position information of cluster centroids. *centers* takes less than 1% of total memory footprint.

Table 2: Benchmark information.

Benchmark	Description	Memory footprint size of two input problems
CG	Iterative solver	Class A: 55MB, Class B: 398MB
FT	Spectral method	Class A: 321MB, Class B: 1283MB
MG	Iterative solver	Class A: 431MB, Class B: 431MB
Kmeans	Clustering analysis	kdd_cup: 133MB, 819200: 222MB

4.3 Recomputability Summary.

We evaluate and characterize application recomputability by triggering crashes at different execution phases and changing the number of threads to run benchmarks. Each case of our study is a combination of using a specific number of threads and triggering crashes at a specific execution phase. Each case of our study includes 100 crash tests. For each crash test, we stop the benchmark in one iteration of the main loop, and then restart the benchmark and examine if the benchmark can complete successfully by running the remaining iterations of the main loop. For the benchmarks CG and MG, the number of iterations of the main loop from one run to another remains constant, but for Kmeans, the number of iterations is not constant from one run to another. In addition, Kmeans can always recompute (converge) successfully after the crash. To evaluate the recomputability of Kmeans, we use the number of iterations after the crash for Kmeans to converge as a metric. If Kmeans needs more iterations to converge after the crash than before the crash, then Kmeans has bad recomputability.

4.4 Recomputability at Different Execution Phases

We trigger crashes at different execution phases of each benchmark. For CG and Kmeans, we evenly divide the whole iteration space of the main loop into three parts, each of which corresponds to one phase. For MG, we evenly divide the whole iteration space into two parts (not three parts), because MG has a small number of iterations. For FT, we only have one iteration, as described in Appendix A. We divide the iteration into four phases (evolve, cfft3, cfft2, and cfft1) based on algorithm knowledge. Our following discussion focuses on the recomputation results of using one thread to run benchmarks. Figures 3a-6b shows the results.

CG with Class A as input shows strong recomputability: at least 96% of all crash tests can recompute successfully. However, when we use a larger input (Class B), CG shows weak recomputability in Phases 2 and 3 (the recomputation success rate is 46% and 7% respectively). FT shows very weak recomputability: the recomputability success rate is less than 6% in all cases. MG also shows relatively weak recomputability: the recomputability success rate is 15%-20%. Kmeans can recompute successfully in all cases, showing strong recomputability.

Observation 1. Different applications have large variance in recomputability. Some applications (e.g., CG and Kmeans) can recompute successfully in almost all cases, while some applications (e.g., FT) have close to zero tolerance to crash inconsistency.

Observation 2. Application recomputability is related to the input problem size. With different input problems, application recomputability can behave differently.

The observation 2 is aligned with our intuition. The application with a larger input problem can lead to a smaller portion of data objects in the cache hierarchy. This reduces the data inconsistent rate when a crash happens, and results in a better possibility to recompute.

When crashes happen at different execution phases of CG (Class B as input), CG shows different recomputability (100%, 46% and 7% for Phases 1, 2 and 3 respectively). CG does not show good recomputability at the late phase (Phase 3). Kmeans shows the similar results: when crashes happen at different execution phases, Kmeans uses a different number of iterations to converge. At the late phase (Phase 3), Kmeans needs a larger number of iterations.

Observation 3. Application recomputability is different across different execution phases.

The iterative structure of some applications has capabilities of tolerating approximate computation by amortizing approximation across iterations [11]. A crash and restart cause approximate computation, because of data inconsistence. A crash happening at the early execution phase has more iterations to tolerate approximation and has a higher possibility to recompute.

Implication 1. If we enforce crash consistency to improve application recomputability, we do not need to enforce it throughout application execution. Reducing the necessity of enforcing crash consistency is helpful to improve application performance. Some applications with specific input problems are naturally recomputable after crashes. They do not need to enforce crash consistency.

4.5 Recomputability with Different Numbers of Threads

We use 1, 4 or 8 threads to run applications. The results are shown in Figures 3a-6b. For CG with Class A, the number of threads has a significant impact on recomputability. As the number of threads increases, the recomputability goes worse. The recomputability for 1, 4 and 8 threads is 99%, 93%, and 45%, respectively. However, for CG (Class B), FT, MG and kmeans, the recomputability is not sensitive to the number of threads.

Observation 5. Application recomputability can be negatively impacted by the number of threads. However, applications with weak recomputability (e.g., FT and MG) remain to have weak recomputability when using different numbers of threads.

We attribute the above observation to the possible larger working set for critical data objects in caches when using a larger number of threads. When a crash happens, having a larger working set size for critical data objects in caches means more data is inconsistent. On the other hand, using a larger number of threads can cause higher data consistent rate, because of more cache line eviction. More cache line eviction implicitly causes more data to be consistent. We discuss the data inconsistent rate in Section 4.6.

Furthermore, Kmeans seems to be a special case. Kmeans has strong recomputability: the recomputation rate is always 100%, no matter how many threads we use to run Kmeans and trigger crashes. Different from CG (Class A) that also has strong recomputability, the recomputability of Kmeans is not impacted by the number of threads at all. We attribute such observation to the strong tolerance to data corruption of Kmeans.

Implication 2. When using the different number of threads, we must use different strategies to ensure application recomputability. When using a larger number of threads to run an application, there is often a need to enforce stronger crash consistency.

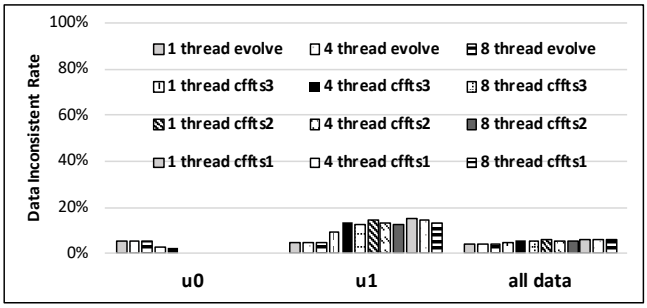
4.6 Analysis based on Data Inconsistent Rate

Figures 7a-10b show the data inconsistent rate for individual critical data objects as well all data objects. We define the data inconsistent rate in Section 3.

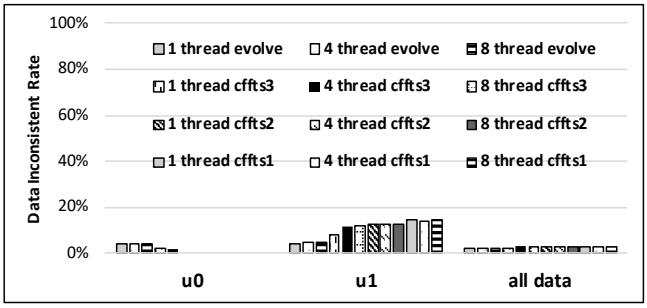
For CG, the data objects q and r have a high inconsistent rate in all cases. The variance of data inconsistent rate across cases is also small. We conclude that the recomputability of CG seems to be less correlated with the data inconsistent rate of these two data objects. We further notice that the data objects p and z has a large variance in data inconsistent rate across cases, when using Class A as input. For p, using a larger number of threads causes higher data inconsistent rate; for z, using a larger number of threads has opposite effects. We conclude that using a larger number of threads, accessing to p and z may have opposite impact on application recomputability. Given the fact that the recomputability of CG (Class A as input) is not sensitive to the number of threads, the impacts of p and z on application recomputability seem to be neutralized.

We have the similar observations for other benchmarks.

Observation 6. We cannot easily explain the variance of application recomputability based on the data inconsistent rate. There seems to be a small correlation between application recomputability and the data inconsistent rate.

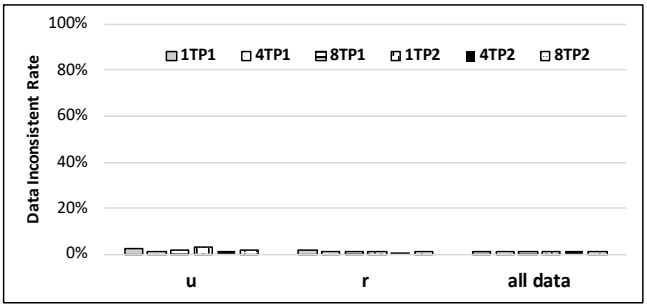


(a) CLASS=A

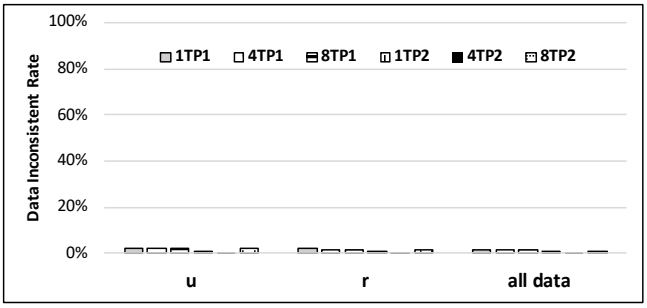


(b) CLASS=B

Figure 8: Data inconsistent rate for FT.



(a) CLASS=A



(b) CLASS=B

Figure 9: Data inconsistent rate for MG. In the legend, "T" stands for thread and "P" stands for phase. xTyP means using x threads and trigger crashes in Phase y.

The above observation may be because of the following reason. The data inconsistence rate only tells us that data is inconsistent, but cannot quantify the value difference between caches and main memory. Two crash tests may cause the same data inconsistent rate for a data object, but have quite different data values in the data object. Different data values can cause different application recomputability. Using the different number of threads and different input problems can cause a big difference in data values between caches and main memory when the crash happens. Such big differences cause different application recomputability.

Implication 3. Considering the cache effects to determine application inconsistent rate is not sufficient to understand application recomputability. We must also consider how different data values in caches and main memory are when the crash happens.

4.7 Discussions and Future Work

NVC is based on PIN that uses binary instrumentation. Running an application with a large memory footprint and intensive memory accesses with NVC can be time-consuming. In our experiments, using NVC to run the application with large input size can cause hundreds of times slowdown. Such long execution time brings challenges for running a large number of crash tests. We plan to extend our work by introducing a new technique to accelerate our analysis. In particular, we plan to proportionally scale down the application's data set and cache size for faster simulation without losing the result correctness for quantifying application recomputability.

NVC allows us to learn the application recomputability and the reason behind. We plan to learn more representative applications and introduce a mechanism to leverage application recomputability to avoid checkpoint or cache flushing for better performance.

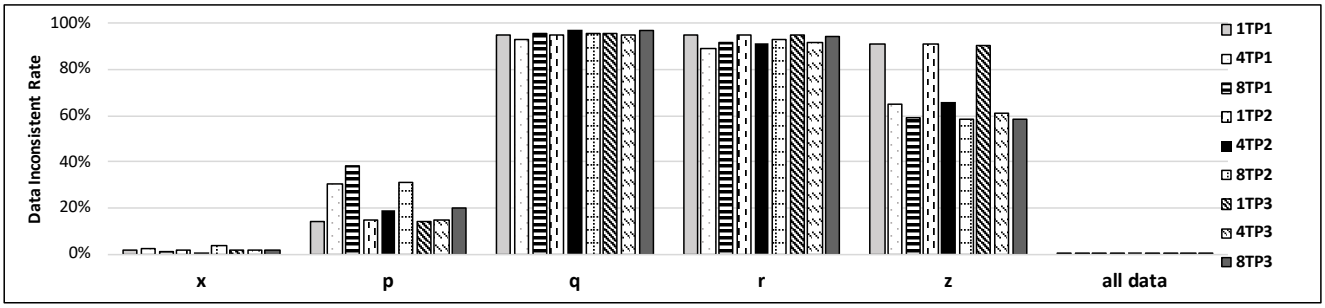
5 RELATED WORK

Many existing work focuses on enabling crashing consistency in NVM, using software- and hardware-based approaches. Different from the existing work, we study application recomputability without crash consistency. We review the existing work related to crash consistency as follows.

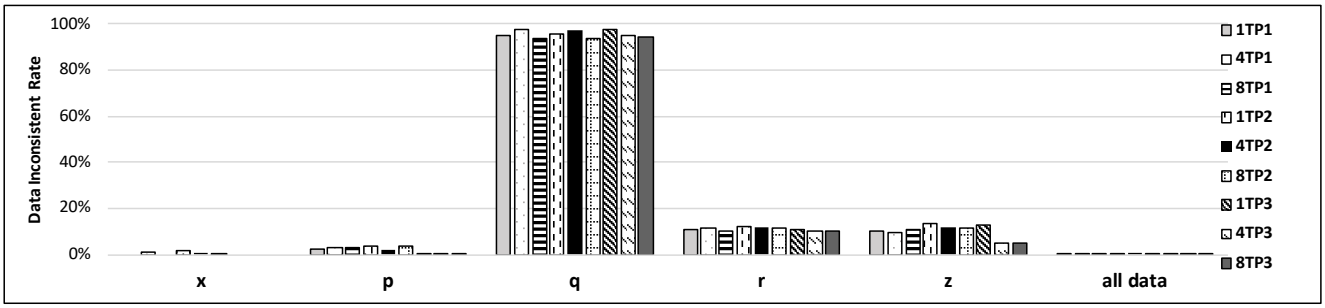
Software support for crash consistency. Enabling crash consistency in NVM with software-based approaches is widely explored. Undo logging and redo logging are two of the most common methods to enable crash consistency, often based on atomic and durable transactions. Using undo and redo logging, once a transaction fails or the application crashes, any uncommitted modifications are ignored, and the application rolls back to the latest version of data in the log.

Persistent Memory Development Kit (PMDK) [13] from Intel supports the transaction system in NVM by undo logging. Similarly, NV-Heaps [7], REWIND [3] and Atlas [2] adopt write-ahead undo logging in NVM. Kolli et al. [15] propose an undo logging that minimizes the write ordering constraint by delaying to commit the data modification.

Mnemosyne [28], a set of programming APIs and libraries for programming with persistent memory, uses redo logging. Lu et al. [17] optimize Mnemosyne to reduce the overhead of supporting transaction by delaying and minimizing the cache flushing. To



(a) CLASS=A



(b) CLASS=B

Figure 7: Data inconsistent rate for CG. In the legend, "T" stands for thread and "P" stands for phase. xTyP means using x threads and trigger crashes in Phase y.

achieve that, they maintain the correct overwrite order of data but do not write them back into memory immediately. A full processor cache flushing will be scheduled when they accumulate enough uncommitted data. Giles et al. [10] provide a redo logging based lightweight atomicity and durability transaction by ensuring fast paths to data in processor caches, DRAM, and persistent memory tiers.

Some existing work focuses on enabling crash consistency for specific data structures, including NV-Tree [32], FPTree [22], NVC-Hashmap [25], CDDS [26] and wBTree [5]. Those data structures support atomic and durable updates, and hence support crash consistency.

Our work can be very helpful for the above work. In particular, NVC can be used to check if crash consistency based on undo log and redo log enables data consistence as expected.

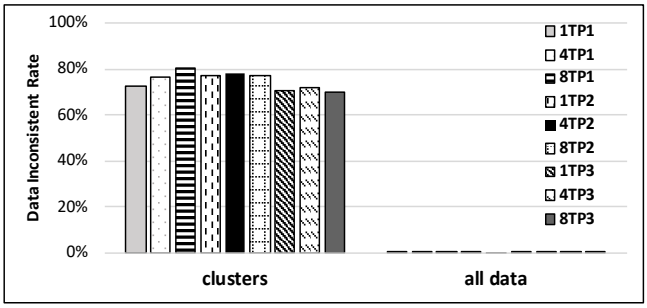
Some existing work considers crash consistency with the context of file system [8, 9, 30]. NOVA [31] is such a file system optimized for heterogeneous memory (DRAM and NVM) systems. It provides strong consistency guarantees and maintains independent logs for each inode to improve scalability. Octopus [16] is another example. Octopus is a RDMA-enabled distributed persistent memory file system. Octopus can have high performance when enforcing metadata consistency, by a "collect-dispatch" transaction. With the collect-dispatch transaction, Octopus collects data from remote servers for local logging and then dispatching them to remote sides by RDMA primitives.

Among the above software-based work, some of them [15, 17] in fact relaxes requirements on crash consistency and does not require crash consistency to be timely enforced, in order to have better performance. Since our work does not require crash consistency, our work also relaxes requirements on crash consistency.

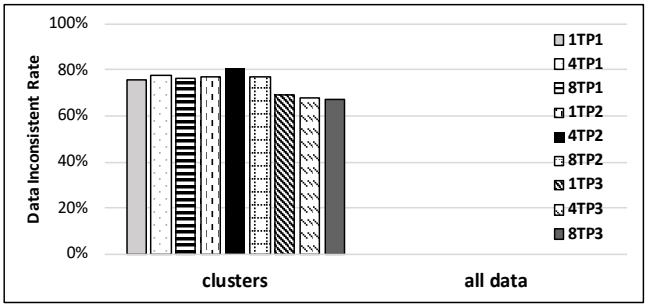
Hardware support for crash consistency. Lu et al. [18] use hardware-based logging mechanisms to relax the write ordering requirements both within a transaction and across multiple transactions. To achieve such goal, they largely modify the cache hierarchy and propose a non-volatile last level CPU cache. Ogleari et al. [20] combine undo and redo hardware logging scheme to relax ordering constraints on both caches and memory controllers for NVM-based systems. Meanwhile, to minimize the frequency of using write-back instructions, they add a hardware-controlled cache to implement a writeback cache. Our work is different from the above hardware-based work, because we do not require hardware modification for crash consistency.

6 CONCLUSIONS

Using NVM as main memory brings an opportunity to leverage NVM's non-volatility for application restarting and recomputing based on the remaining data objects in NVM, after the application crashes. Different from the existing work that enables crash consistency for application recomputation, we statistically quantify recomputability of a set of applications without crash consistency in NVM. We develop a tool (named NVC) that allows us to trigger random crash, examine data consistency and restart application for our study. Using the tool, we real that some applications have very good recomputability without crash consistency on critical data objects. Our work is the first one that studies application recomputability without crash consistency. Our work opens a door to remove runtime overhead of those crash-consistency mechanisms



(a) kdd_cup as input



(b) 819200 as input

Figure 10: Data inconsistent rate for kmeans. In the legend, "T" stands for thread and "P" stands for phase. xTyP means using x threads and trigger crashes in Phase y.

(e.g., logging and checkpoint). Our work makes NVM a more feasible solution for application recomputation in those fields with the high-performance requirement.

REFERENCES

[1] Milind Chabbi, Xu Liu, and John Mellor-Crummey. 2014. Call Paths for Pin Tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*.

[2] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*.

[3] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*.

[5] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797.

[6] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura Grupp, Rajesh Gupta, Ranjit Jhala, and Steve Swanson. 2011. NV-heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proc. of 16th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*.

[9] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.

[10] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*.

[11] Serge Gratton, Philippe L. Toint, and Anke Tröltzsch. 2011. How much gradient noise does a gradient-based linesearch method tolerate?

[12] Xiaochen Guo, Engin Ipek, and Tolga Soyata. 2010. Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)*.

[13] Intel. 2014. Persistent Memory Development Kit. https://pmem.io/. (2014).

[14] Intel. 2014. Intel NVM Library. http://pmem.io/nvml/libpmem/. (2014).

[15] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[16] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*.

[17] Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred Persistence: Efficient Transactions in Persistent Memory. *Trans. Storage* 12, 1, Article 3 (Jan. 2016), 29 pages.

[18] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*.

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. 2005 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '05)*. Chicago, IL, 190–200.

[20] Ethan L. Miller Matheus Ogleari and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[21] P. Mazumder, S. M. Kang, and R. Waser. 2012. Memristors: Devices, Models, and Applications [Scanning the Issue]. *Proc. IEEE* (2012), 1911–1919.

[22] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*.

[23] Andy Rudoff. 2013. Programming Models for Emerging Non-Volatile Memory Technologies. *The USENIX Magazine* 38, 3 (2013), 40–45.

[24] Arthur Sainio. 2016. NVDIMM: Changes are Here So WhatâĂŹs Next?. In *In-Memory Computing Summit 2016*.

[25] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics (IMDM '15)*.

[26] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*.

[27] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proc. 16th Annu. Middleware Conference (Middleware '15)*. Vancouver, Canada, 37–49.

[28] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[29] H. Volos, A. J. Tack, and M. M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[30] X. Wu and A. L. N. Reddy. 2011. SCMFS: A file system for Storage Class Memory. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[31] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*.

[32] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*.

[33] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai. 2017. Algorithm-Directed Crash Consistence in Non-volatile Memory for HPC. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*.

A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System

Prasanth Chatarasi and Vivek Sarkar
Georgia Institute of Technology
Atlanta, Georgia, USA
{cprasanth,vsarkar}@gatech.edu

ABSTRACT

Unlike dense linear algebra applications, graph applications typically suffer from poor performance because of 1) inefficient utilization of memory systems through random memory accesses to graph data, and 2) overhead of executing atomic operations. Hence, there is a rapid growth in improving both software and hardware platforms to address the above challenges. One such improvement in the hardware platform is a realization of the Emu system, a thread migratory and near-memory processor. In the Emu system, a thread responsible for computation on a datum is automatically migrated over to a node where the data resides without any intervention from the programmer. The idea of thread migrations is very well suited to graph applications as memory accesses of the applications are irregular. However, thread migrations can hurt the performance of graph applications if overhead from the migrations dominates benefits achieved through the migrations.

In this preliminary study, we explore two high-level compiler optimizations, i.e., loop fusion and edge flipping, and one low-level compiler transformation leveraging hardware support for remote atomic updates to address overheads arising from thread migration, creation, synchronization, and atomic operations. We performed a preliminary evaluation of these compiler transformations by manually applying them on three graph applications over a set of RMAT graphs from Graph500.—Conductance, Bellman-Ford's algorithm for the single-source shortest path problem, and Triangle Counting. Our evaluation targeted a single node of the Emu hardware prototype, and has shown an overall geometric mean reduction of 22.08% in thread migrations.

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286481

KEYWORDS

Loop fusion, Edge flipping, Graph algorithms, Thread migratory, Near-memory, Atomic operations, The Emu system, Compilers

ACM Reference Format:
Prasanth Chatarasi and Vivek Sarkar. 2018. A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3286475.3286481

1 INTRODUCTION

Though graph applications are increasing in importance with the advent of "big data", achieving high performance with graph algorithms is non-trivial and requires careful attention from programmers [20]. Two significant bottlenecks to achieving higher performance on existing CPU and GPU-based architectures are 1) inefficient utilization of memory systems through random memory accesses to graph data, and 2) overhead of executing atomic operations. Since graph applications are typically cache-unfriendly and are not well supported by existing traditional architectures, there is growing attention being paid by the architecture community to innovate suitable architectures for such applications. One such innovation is the Emu system, a highly scalable near memory system with support for migrating threads without programmer intervention [8]. The system is designed to improve the performance of data-intensive applications exhibiting weak locality, i.e., from irregular and cache-unfriendly memory access which are often found in graph analytics [18] and sparse matrix algebra operations [19].

Emu architecture. An Emu system consists of multiple Emu nodes interconnected by a fast rapid IO network, and each node (shown in Figure 1) contains *nodelets*, stationary cores and migration engines. Each *nodelet* consists of a Narrow Channel DRAM (NCDRAM) memory unit and multiple Gossamer cores, and the co-location of the memory unit with the cores makes the overall Emu system a near-memory system. Even though each nodelet has a different physical co-located memory unit, the Emu system provides a logical view

of the entire memory via the partitioned global address space (PGAS) model with memory contributed by each nodelet.

Each gossamer core of a nodelet is a general-purpose, simple pipelined processor with no support for data caches and branch prediction units, and is also capable of supporting 64 concurrent threads using fine-grain multi-threading. A key aspect of the Emu system is thread migration by hardware, i.e., a thread is migrated on a remote memory read by removing thread context from the nodelet and transmitting the thread context to a remote nodelet without programmer intervention. As a result, each nodelet requires multiple queues such as service, migration and run queues to process threads spawned locally (using *spawn* instruction) and also migrated threads.

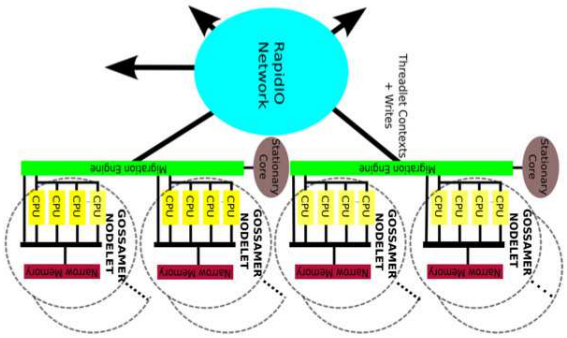


Figure 1: Overview of a single Emu node (Figure source: [10]), where a dotted circle represents a nodelet. Note that, the co-location of the narrow channel memory unit (NCDRAM) with gossamer cores makes the overall Emu system a near memory system.

Software support. The Emu system supports the Cilk parallel programming model for thread spawning and synchronization using `cilk_spawn`, `cilk_sync` and `cilk_for` constructs [11]. Since the Emu hardware automatically takes care of thread migration and management; hence the Cilk runtime is discarded in the toolchain. Also, it is important to note that appending a `cilk_spawn` keyword before a function invocation to launch a new task is directly translated to the `spawn` instruction of the Emu ISA during the compilation. The Emu system also provides libraries for data allocation and distribution over multiple nodelets, and intrinsic functions for atomic operations and migrating thread control functions. Also, there has been significant progress made in supporting standard C libraries on the Emu system.

Even though the Emu system is designed to improve the performance of data-sensitive workloads exhibiting weak-locality, the thread migrations across nodelets can hamper the performance if overhead from the thread migration dominates the benefits achieved through the migration. In the next section, we study both high-level and low-level compiler transformations which can be applied to original graph applications to mitigate the overheads as mentioned earlier.

2 COMPILER TRANSFORMATIONS

In this section, we discuss two high-level compiler transformations (*Node fusion* and *Edge flipping*)[1], and one low-level compiler transformation leveraging the *remote atomic update* feature of the hardware, to mitigate the impact of overheads in the performance of graph applications on the Emu system.

2.1 Node/Loop Fusion

Programmers write graph applications with multiple parallel loops over nodes of a graph either to 1) compute various properties of a node (e.g., in *Conductance* [7, 21]), or 2) query on computed properties of nodes (e.g., in *Average teenage followers* [17]). In such scenarios, multiple such parallel loops can be grouped into a single parallel loop, and compute multiple properties in the same loop or query immediately after computing the properties. This grouping can result in reducing thread migrations occurring in later loops, and also overheads arising from thread creation and synchronization. The grouping of multiple such parallel loops is akin to loop fusion, a classical compiler transformation for improving locality; but we use the transformation to reduce unnecessary migrations (for more details, see Section 3.2).

2.2 Edge Flipping

Edge flipping is another compiler transformation discussed in [15] to flip a loop over incoming edges of a node with outgoing edges of the node. However, we generalize the edge flipping transformation to allow flips between both incoming and outgoing edges. To allow this bi-directional flipping, the transformation assumes an input graph to be bi-directional, i.e., each node in the graph stores a list of incoming edges along with outgoing edges.

Vertex centric graph algorithms such as Page rank, Bellman-Ford algorithm for single-source shortest path, Page coloring offer opportunities to explore the edge flipping transformation since these algorithms either explore incoming edges of a node to avoid synchronization (pull-based approach), or explore outgoing edges to reduce random memory accesses (push-based approach), or explore a combination [6, 29]. We discuss the above push-pull dichotomy in Section 2.2, using

[1]Note that these high-level transformations – node fusion and edge flipping – have already been explored in past work on optimizing graph algorithms on the x86 architectures [15], and we are evaluating them in the context of the EMU system in this paper.

the Bellman-Ford's algorithm as a representative of vertex-centric graph algorithms.

2.3 Use of Remote Updates

Remote updates, one of the architectural features of the Emu, are stores and atomic operations to a remote memory location that don't require returning a value to the thread, and these operations do not result in thread migrations [8]; instead they send an information packet to the remote nodelet containing the data and the operation to be performed. These remote updates also can be viewed as very efficient special-purpose migrating threads, and they don't return a result unlike regular atomic operations, but they return an acknowledgement that the memory unit of the remote nodelet has accepted the operation. We leverage this feature as a low-level compiler transformation replacing regular atomic operations that don't require returning a value by the corresponding remote updates. The benefits of this transformation can be immediately seen in vertex-centric algorithms (Section 3.3) and also in the triangular counting algorithm (Section 3.4).

3 EXPERIMENTS

In this section, we present the benefits of applying the compiler transformations on graph algorithms. We begin with an overview of the experimental setup and the graph algorithms used in the evaluation, and then we present our discussion on preliminary results for each algorithm.

3.1 Experimental Setup

Our evaluation uses dedicated access to a single node of the Emu system, i.e., the Emu Chick prototype[2] which uses an Arria 10 FPGA to implement Gossamer cores, migration engines, and stationary cores of each nodelet. Table 1 lists the hardware specifications of a single node of the Emu Chick.

Table 1: Specifications of a single node of the Emu system.

	Emu system
Microarch	Emu1 Chick
Clock speed	150 MHz
#Nodelets	8
#Cores/Nodelet	1
#Threads/Core	64
Memorysize/Nodelet	8 GB
NCDRAM speed	1600MHz
Compiler toolchain	emusim.HW.x (18.08.1)

In the following experiments, we compare two experimental variants: 1) Original version of a graph algorithm running with all cores of a single node and 2) Transformed version after manually applying compiler transformations on the graph algorithm. In all experiments, we measure only the execution time of the kernel and report the geometric mean execution time measured over 50 runs repeated in the same environment for each data point. The speedup is defined as the execution time of the original version of a graph algorithm divided by the execution time of the transformed version of the program running with all cores of a single node of the Emu system in both cases,i.e., eight cores.

We also use an in-house simulation environment of the Emu prototype, whose specifications match with the hardware details mentioned in Table 1, to measure statistics of programs such as thread migrations, threads created and terminated. We are not currently aware of any methods for extracting these statistics from the hardware. We define the percentage reduction in thread migrations[3] as follows:

$$\%\text{reduction in migrations}$$

$$= (1 - (\frac{\#\text{migrations in the transformed version}}{\#\text{migrations in the original version}})) \times 100$$

Finally, we evaluate the benefits of compiler transformations by measuring both improvements in execution time on the Emu hardware and reduction in thread migrations on the Emu simulator.

Graph applications: For our evaluation, we consider three graph algorithms, i.e., 1) Conductance algorithm, 2) Bellman-Ford's algorithm for Single-source shortest path (SSSP) problem, and 3) Triangle counting algorithm. Both original and transformed versions of above algorithms are implemented using the Meatbee framework [13], an in-house experimental streaming graph engine used to develop graph algorithms for the Emu system. The Meatbee framework, inspired by the STINGER framework [9], uses a striped array of pointers to distribute the vertex array across all nodelets in the system, and also implements the adjacency list as a hash table with a small number of buckets.

Input data-sets: We use RMAT graphs (edges of these graphs are generated randomly with a power-law distribution), scale[4] from 6 to 14 as specified by Graph500 [2]. Note that all the above graphs specified by Graph500 are generated using the utilities present in the STINGER framework. Table 2 presents details of the RMAT graphs used in our evaluation, and total thread migrations and execution times of the original graph algorithms on the Emu system.

[2]Several aspects of the system are scaled down in the prototype Emu system, e.g., number of gossamer cores of a nodelet

[3]Note that the thread migration counts are for the entire program, and we are not currently aware of any existing approaches on how to obtain migration counts for a specific region of code.

[4]A scale of n for an input graph refers to having 2^n vertices.

Scale	#vertices	#edges	Thread migrations in the original program			Execution time of the original program (ms), geometric mean of 50 runs		
			Conductance	SSSP-BF	Triangle counting	Conductance	SSSP-BF	Triangle counting
6	64	1K	6938	10915	26407	4.45	26.32	53.63
7	128	2K	13812	22851	84168	7.51	393.04	163.36
8	256	4K	28221	48354	252440	13.89	1634.64	547.84
9	512	8K	59068	104653	809423	32.13	2887.61	1694.09
10	1K	16K	122088	220204	2475350	64.59	4589.42	3942.55
11	2K	32K	253364	474118	7381977	134.43	10225.10	12649.30
12	4K	64K	522530	1136600	21777902	844.38	32140.30	36199.60
13	8K	128K	1065640	2332741	64063958	1841.53	-	185864.00
14	16K	256K	2171311	4569519	180988114	7876.99	-	721578.00

Table 2: Experimental evaluation of three graph algorithms (Conductance, SSSP-BF and Triangle counting) on the RMAT graphs from scales 6 to 14 specified by Graph500. Transformations applied on the algorithms: Conductance/SSSP-BF/Triangle counting: (Node fusion)/(Edge flipping and Remote updates)/ (Remote updates). The evaluation is done a single node of the Emu system described in Table 1. Note that we had intermittent termination issues while running SSSP-BF from scale 13-14 on the Emu node, and hence we omitted its results.

3.2 Conductance algorithm

The conductance algorithm is a graph metric application to evaluate a graph partition by counting the number of edges between nodes in a given partition and nodes in other graph partitions [7, 21]. The algorithm is frequently used to detect community structures in social graphs. An implementation of the conductance algorithm is shown in Algorithm 1. The implementation[5] at a high-level consists of three parallel loops iterating over vertices of a graph to compute different properties (such as `din, dout, dcross`) of a given partition (specified as `id` in the algorithm). Finally, these properties are used to compute conductance value of the partition of the graph.

As can be seen from the implementation, the EMU hardware spawns a thread for every vertex (v) of the graph from the first parallel loop (lines 2-4), and migrates to a nodelet where the vertex property `partition_id` is stored after encountering the property (`v.partition_id`) at line 3. Since the `degree` property of the vertex (v) is also stored on the same nodelet as of the other property[6], the thread doesn't migrate on encountering the property, `v.degree`, at line 4. After reduction of the `din` variable, the hardware performs a global synchronization of all spawned threads because of an implicit barrier after the parallel loop. After the synchronization, the hardware again spawns a thread for every vertex from the second parallel loop (lines 5-7), and migrates after encountering the same property (`v.partition_id` at line 6). The same behavior is repeated in the third parallel loop as well (lines 8-12). The repeated migrations to the same nodelet from multiple parallel loops, which arise from accessing the same property or a different property which is stored on the same nodelet, can be reduced by fusing all the three parallel loops into a single loop. Also, the fusion of multiple parallel loops can reduce the overhead of multiple thread creations and synchronization. As can be seen from Figure 2, we have observed a geometric mean reduction of 6.06% in the total number of thread migrations after fusing three loops. As a result, we also found a geometric mean speedup of 1.95x in the execution time of the computation over the scale 6-14 of RMAT graphs specified by Graph500. This performance improvement demonstrates the need for fusing parallel loops over nodes of a graph to compute values/properties together

Algorithm 1: An implementation of the Conductance algorithm [7, 21].

```
1  def CONDUCTANCE(V, id):
2  for each v ∈ V do in parallel with reduction
3      if v.partition_id == id then
                    ▷ Thread migration for v.partition_id value
4          din+ = v.degree
5  for each v ∈ V do in parallel with reduction
6      if v.partition_id != id then
7          dout+ = v.degree
8  for each v ∈ V do in parallel with reduction
9      if v.partition_id == id then
10         for each nbr ∈ v.nbrs do
11             if nbr.partition_id != id then
12                 dcross+ = 1
13 return dcross/((din < dout)?din : dout)
```

[5]The implementation is from a naive translation from existing graph analytics domain-specific compilers for non-EMU platforms.

[6]The properties of vertices (such as `partition_id`, `degree`) are allocated similar to the vertex allocation, i.e., uniformly across all nodelets.

to reduce thread migrations in applications such as Conductance.

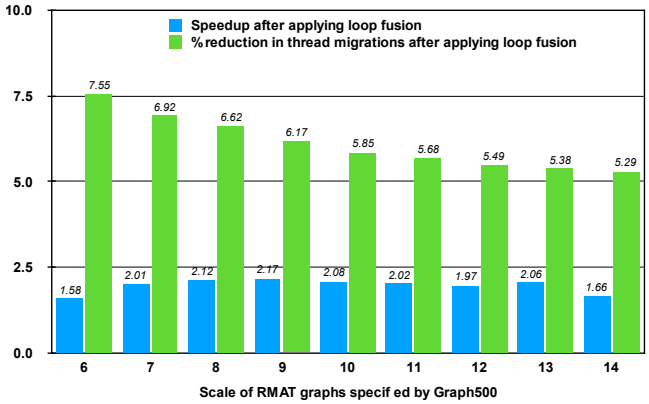


Figure 2: Speedup over the original conductance algorithm on a single Emu node (8 nodelets) and % reductions in thread migrations after applying loop fusion.

3.3 Single Source Shortest Path using Bellman-Ford's Algorithm (SSSP-BF)

Bellman-Ford's algorithm is used to compute shortest paths from a single source vertex to all the other vertices in a weighted directed graph. An implementation of the algorithm is shown in Algorithm 2. We added a minor step (at lines 15, 18, 23-25) in the body of the `t-loop` to the implementation for termination if subsequent iterations of the `t-loop` will not make any more changes, i.e., the distance computed (`temp_distance`) for each vertex in the current iteration is the same as the distance in the previous iteration (`distance`).

As can be seen from the implementation, the EMU hardware spawns a thread for every vertex (v) of the graph from the parallel loop (lines 6-13) nested inside the `t-loop`. The thread responsible for a particular vertex (v) in a given iteration (t) migrates to an incoming neighbor vertex (u) on encountering the accesses `distance(`u`)` and `weight(u, v)` (line 8). After adding the values, the thread migrates back to the original node for writing after encountering the access `temp_distance(u)` (line 9). The same migration behavior is repeated for every incoming neighbor vertex, and finally the local value based on the best distance from incoming neighbors is computed. This approach is commonly known as a pull-based approach since the vertex pulls information from incoming neighbors to update its local value. However, the back and forth migrations for every neighbor vertex via incoming edges can be avoided by doing the edge flipping transformation (discussed in Section 2.2), i.e., the loop iterating over incoming edges is flipped into a loop over outgoing edges. The transformations leads to a push-based approach for the SSSP algorithm, in which a vertex pushes its contribution (`distance(u) + weight(u, v)`) to its neighbors accessible via outgoing edges and doesn't require migrating to the neighbors, as in the pull-based approach. Since multiple vertices can have a common neighbor, the contribution is done atomically, i.e., by using atomic_min in our implementation.

As a result of applying edge-flipping transformation, we have observed a geometric mean reduction of 8.69% in the total number of thread migrations (shown in Figure 3). However, the push-based approach with regular atomic updates didn't perform well compared with the pull-based approach from the scale of 7 to 9 (shown in Figure 4), because of irregularity in the input graphs and imbalance in the number of incoming and outgoing edges. As a result, the cost of migrating back and forth in the pull-based approach was not expensive compared to doing more atomic updates in the push-based approach for the above data points. This observation is in accordance with the push-pull dichotomy discussed in [6, 29].

Algorithm 2: An implementation of the Bellman-Ford's algorithm (SSSP-BF).

```
1  def SSSP_BFS(V, id):
2  distance(id) ← 0
3  distance(v) ← MAX, for ∀v ∈ (V − {id})
4  temp_distance(v) ← 0, for ∀v ∈ V
5  for t ← 0 to |V| − 1 do
6      for each v ∈ V do in parallel
7          for each u ∈ incoming_neighbors(v) do
8              temp = distance(u) + weight(u, v)
                                  ▷ Migration for distance(u) value
9              if distance(v) > temp then
10                 temp_distance(v) = temp
11             end
12         end
13     endfor
14
15     modified ← false
16     for each v ∈ V do in parallel
17         if distance(v)! = temp_distance(v) then
18             modified ← true
19             distance(v) = temp_distance(v)
20         end
21     endfor
22
23     if modified == false then
24         break;
25     end
26 end
27 return distance;
```

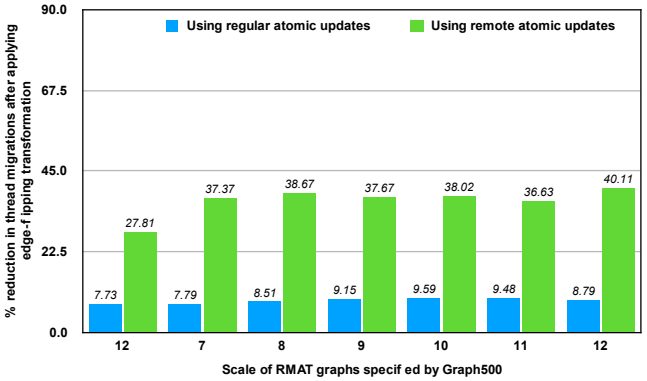


Figure 3: % reductions in thread migrations of SSSP-BF algorithm after applying edge flipping with regular atomic updates and with remote atomic updates on a single node (8 nodelets) of Emu Chick.

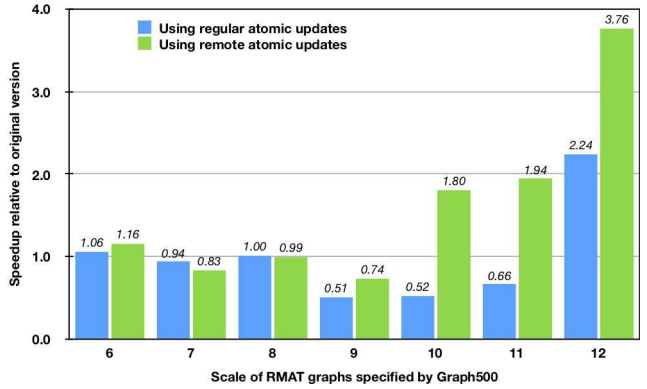


Figure 4: Speedup of SSSP-BF algorithm on a single Emu node (8 nodelets) after applying edge flipping with regular atomic updates and with remote updates.

Furthermore, the push-based approach can be strengthened by replacing regular atomic updates with remote atomic updates since a node which is pushing its contribution (i.e., its distance) to neighbors via outgoing edges doesn't need a return value. By doing so, we have observed a geometric mean reduction of 30.28% in thread migrations (shown in Figure 3) compared to the push-based approach with regular atomic updates. Also, there is an overall geometric mean improvement of 1.57x in execution time relative to the push-based approach with regular atomic updates (shown in Figure 4). The above performance improvement demonstrates the need for using remote atomic updates for scalable performance, and also exploring hybrid approaches involving both push and pull strategies based on input graph data.

3.4 Triangle Counting Algorithm

Triangle counting is another graph metric algorithm which computes the number of triangles in a given undirected graph, and also computes the number of triangles that each node belongs to [24]. The algorithm is frequently used in complex network analysis, random graph models, and also real-world applications such as spam detection. An implementation of the Triangle counting is shown in Algorithm 3, and it works by iterating over each vertex(v), picking two distinct neighbors (u, w), and check if there exists an edge between them to be part of a triangle. Also, the implementation avoids duplicate counting by delegating the counting of a triangle to the vertex with lower id.

Algorithm 3: An implementation of the Triangle counting algorithm [24].

```
1  tc(v) ← 0, for ∀v ∈ (V)
2  for each v ∈ V do in parallel
3      for each u ∈ v.nbrs do
4          if nbr1 > v then
5              for each w ∈ v.nbrs do
6                  if w > u then
7                      if edge_exists(u, w) then
8                          tc_count ++; //Atomic
9                          tc(v) ++; //Atomic
10                         tc(u) ++; //Atomic
11                         tc(w) ++; //Atomic
                               ▷ Above regular atomics can be
                           replaced by the remote updates.
```

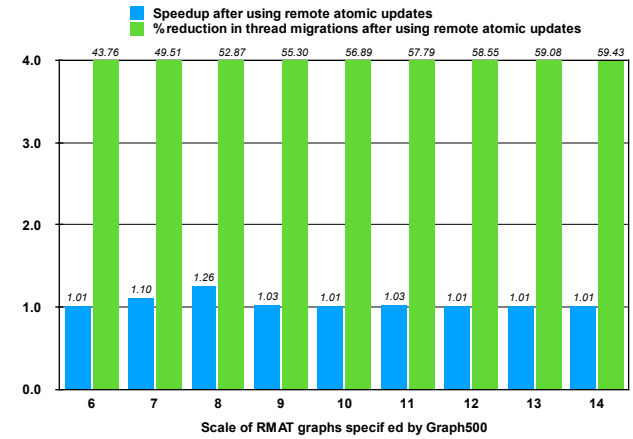


Figure 5: Speedup over the original triangle counting implementation on a single Emu node (8 nodelets) and % reductions in thread migrations after using remote atomic updates.

In the above implementation of the triangle counting algorithm, whenever a triangle is identified (line 7), the implementation atomically increments the overall triangles count and triangle counts of the three vertices of the triangle. As part of the atomic update operation, the thread performs a migration to the nodelet having the address. However, the thread incrementing the triangle counts doesn't need the return value of the increment for further computation; hence, the regular atomic updates can be replaced by remote atomic updates to reduce thread migrations. After replacing with remote updates, we have observed a geometric mean reduction of 54.55% in the total number of thread migrations (shown in Figure 5). As a result, we also found a geometric mean speedup of 1.05x[7] in the execution time of the kernel over the scale 6-14 of RMAT graphs specified by Graph500.

4 RELATED WORK

There is an extensive body of literature in optimizing graph applications for a variety of traditional architectures [3, 4, 27], accelerators [12, 16], and processing in memory (PIM) architectures [1, 22]. Also, there has been significant research done on optimizing task-parallel programs to reduce the overheads arising from task creation, synchronization [23, 25, 30] and migrations [26]. In this section, we discuss past work closely related to optimizing irregular applications for the Emu system and also past work on compiler optimizations in mitigating task (thread) creation, synchronization and migration overhead.

Emu related past work. Kogge et al. in [19] discussed migrating thread paradigm of the Emu system as an excellent match for systems with significant near-memory processing, and evaluated its advantage over a sparse matrix application (SpMV) and a streaming graph analytic benchmark (Firehose). Hein et al. [14] characterized the Emu chick hardware prototype (same as what we used in our evaluation) using custom kernels and discussed memory allocation, thread migrations strategies for SpMV kernels. In this work, we study high-level, and low-level compiler transformations that can benefit existing graph algorithms by leveraging the intricacies discussed in [5, 8, 14, 19, 28].

Programming models support and compiler optimizations for reducing thread creation, synchronization and migration overheads. Task-parallel programs often result in considerable overheads in task creation and synchronization, and hence approaches in [23, 25, 30] presented compiler frameworks to transform the input program to reduce the overheads using optimizations such as task fusion, task chunking, synchronization (`finish` construct) elimination. Our study on the loop fusion transformation to reduce thread creation and synchronization overheads on the Emu system is inspired by the above compiler frameworks and also from the Green-Marl DSL compiler [15].

[7]Note that the computational complexity of the triangle counting algorithm is significant, i.e., $O(m^{\frac{3}{2}})$ where m is number of edges, and even 5% improvement is equivalent to few thousands of msecs as reported in Table 2.

5 CONCLUSIONS AND FUTURE WORK

Graph applications are increasing in popularity with the advent of "big data", but achieving higher performance is not trivial. The major bottlenecks in graph applications are 1) inefficient utilization of memory subsystems through random memory accesses to the graph data, and 2) overhead of executing atomic operations. Since these graph applications are cache-unfriendly and are not well handled by existing traditional architectures, there is growing attention in the architecture community to innovate suitable architectures for such applications.

One of the innovative architecture to handle graph applications is a thread migratory architecture (Emu system), where a thread responsible for computation on a data is migrated over to a nodelet where the data resides. However, there are significant challenges which need to be addressed to gain the potential of Emu system, and they are reducing thread migration, creation, synchronization, and atomic operation overheads. In this study, we explored two high-level compiler optimizations, i.e., loop fusion and edge flipping, and one low-level compiler transformation leveraging remote atomic updates to address the above challenges. We performed a preliminary evaluation of these compiler transformations by manually applying them on three graph applications over a set of RMAT graphs from Graph500.—Conductance, Bellman-Ford's algorithm for the single-source shortest path problem, and Triangle Counting. Our evaluation targeted a single node of the Emu hardware prototype, and has shown an overall geometric mean reduction of 22.08% in thread migrations. This preliminary study clear motivates us in exploring the implementation of automatic compiler transformations to alleviate the overheads arising from running graph applications on the Emu system.

6 ACKNOWLEDGMENTS

We would like to thank Eric Hein for his help in getting us started with the Emu system and using the Meatbee framework to develop graph algorithms. Also, we would like to acknowledge Jeffrey Young for his help with the Emu machine at Georgia Tech.

REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117. https://doi.org/10.1145/2749469.2750386

[2] David A Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E Jason Riedy, and Jeremiah Willcock. 2011. *Graph500 Benchmark 1 (search) Version 1.2*. Technical Report. Graph500 Steering Committee.

[3] David A. Bader and Kamesh Madduri. 2005. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC'05)*. Springer-Verlag, Berlin, Heidelberg, 465–476. https://doi.org/10.1007/11602569_48

[4] D. A. Bader and K. Madduri. 2006. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In *2006 International Conference on Parallel Processing (ICPP'06)*. 523–530. https://doi.org/10.1109/ICPP.2006.34

[5] Mehmet E. Belviranli, Seyong Lee, and Jeffrey S. Vetter. 2018. Designing Algorithms for the EMU Migrating-threads-based Algorithms. In *High Performance Extreme Computing Conference (HPEC), 2018 IEEE*. IEEE.

[6] Maciej Besta, MichałPodstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, New York, NY, USA, 93–104. https://doi.org/10.1145/3078597.3078616

[7] Bela Bollobas. 1998. *Modern Graph Theory*. Springer.

[8] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 2–9. https://doi.org/10.1109/IA3.2016.007

[9] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[10] EmuTechnology. 2017 (accessed December 12, 2017). *Emu System Level Architecture*. http://www.emutechnology.com/products/#lightbox/0/

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

[12] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC'07)*. Springer-Verlag, Berlin, Heidelberg, 197–208.

[13] Eric Hein. 2017. Meatbee, An Experimental Streaming Graph Engine. https://github.gatech.edu/ehein6/meatbee.

[14] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, and J. Riedy. 2018. An Initial Characterization of the Emu Chick. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 579–588. https://doi.org/10.1109/IPDPSW.2018.00097

[15] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 349–362. https://doi.org/10.1145/2150976.2151013

[16] S. Hong, T. Oguntebi, and K. Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 78–88. https://doi.org/10.1109/PACT.2011.14

[17] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. 2014. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 208, 11 pages. https://doi.org/10.1145/2544137.2544162

[18] P. M. Kogge. 2017. Graph Analytics: Complexity, Scalability, and Architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1039–1047. https://doi.org/10.1109/IPDPSW.2017.176

[19] Peter M. Kogge and Shannon K. Kuntz. 2017. A Case for Migrating Execution for Irregular Applications. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms (IA3'17)*. ACM, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/3149704.3149770

[20] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2016. Parallel Graph Analytics. *Commun. ACM* 59, 5 (April 2016), 78–87. https://doi.org/10.1145/2901919

[21] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1, Article 1 (July 2016), 20 pages. https://doi.org/10.1145/2898361

[22] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468. https://doi.org/10.1109/HPCA.2017.54

[23] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 3 (April 2013), 48 pages. https://doi.org/10.1145/2450136.2450138

[24] Thomas Schank. 2007. *Algorithmic Aspects of Triangle-Based Network Analysis*. Ph.D. Dissertation. Universität Karlsruhe.

[25] Jun Shirako, Jisheng M. Zhao, V. Krishna Nandivada, and Vivek N. Sarkar. 2009. Chunking Parallel Loops in the Presence of Synchronization. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/1542275.1542304

[26] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2010. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing (LCPC'09)*. Springer-Verlag, Berlin, Heidelberg, 172–187. https://doi.org/10.1007/978-3-642-13374-9_12

[27] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, Washington, DC, USA, 25–. https://doi.org/10.1109/SC.2005.4

[28] Jeffrey Young, Eric Hein, Srinivas Eswar, Patrick Lavin, Jiajia Li, Jason Riedy, Richard Vuduc, and Tom Conte. 2018. A Microbenchmark Characterization of the Emu Chick. arXiv:cs.DC/1809.07696

[29] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt - A High-Performance DSL for Graph Analytics. *CoRR* abs/1805.00923 (2018). arXiv:1805.00923 http://arxiv.org/abs/1805.00923

[30] Jisheng Zhao, Jun Shirako, V. Krishna Nandivada, and Vivek Sarkar. 2010. Reducing Task Creation and Termination Overhead in Explicitly Parallel Programs. In *Proceedings of the 19th International Conference on PACT*. ACM, New York, NY, USA, 169–180.

Data Placement Optimization in GPU Memory Hierarchy using Predictive Modeling

Larisa Stoltzfus
University of Edinburgh
Edinburgh, UK
larisa.stoltzfus@ed.ac.uk

Murali Emani
Lawrence Livermore National Laboratory
Livermore, CA
emani1@llnl.gov

Pei-Hung Lin
Lawrence Livermore National Laboratory
Livermore, CA
lin32@llnl.gov

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, CA
liao6@llnl.gov

ABSTRACT

Modern supercomputers often use Graphic Processing Units (or GPUs) to meet the ever-growing demands for high performance computing. GPUs typically have a complex memory architecture with various types of memories and caches, such as global memory, shared memory, constant memory, and texture memory. The placement of data on these memories has a tremendous impact on the performance of the HPC applications and identifying the optimal placement location is non-trivial.

In this paper, we propose a machine learning-based approach to build a classifier to determine the best class of GPU memory that will minimize GPU kernel execution time. This approach utilizes a set of performance counters obtained from profiling runs along with hardware features to generate the trained model. We evaluate our approach on several generations of NVIDIA GPUs, including Kepler, Maxwell, Pascal, and Volta on a set of benchmarks. The results show that the trained model achieves prediction accuracy over 90% and given a global version, the classifier can accurately determine which data placement variant would yield the best performance.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies → Machine learning**;

KEYWORDS

GPU, Data placement, Memory, Machine Learning

ACM Reference Format:
Larisa Stoltzfus, Murali Emani, Pei-Hung Lin, and Chunhua Liao. 2018. Data Placement Optimization in GPU Memory Hierarchy using Predictive Modeling. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3286475.3286482

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286482

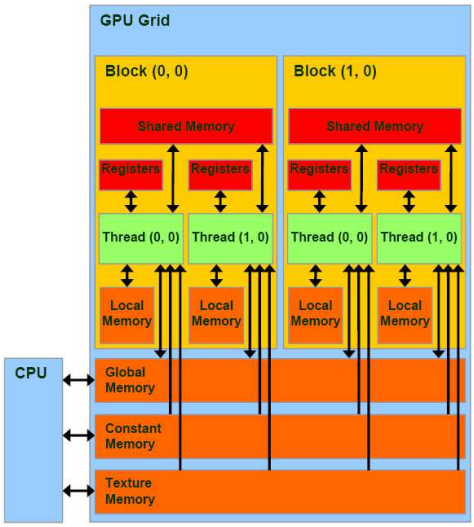


Figure 1: GPU memory hierarchy (from NVIDIA [13])

1 INTRODUCTION

Supercomputers use various types of memory and increasingly complex designs to meet the ever growing need for data by modern massively parallel processors (e.g., Graphic Processing Units or GPUs). On an NVIDIA Kepler GPU, for instance, there are more than eight types of memory (global, texture, shared, constant, various caches, etc.). Figure 1 shows a typical GPU memory hierarchy. With unified CPU-GPU memory space support on the latest GPUs (e.g., NVIDIA Volta), data placement becomes even more flexible, allowing direct accesses to data across the boundaries of multiple GPUs and CPU. This problem will critically affect the effective adoptions of new generations of supercomputers, such as Sierra hosted at LLNL [1] and Summit at ORNL [2] featuring NVIDIA Volta GPUs using 3D stacked memory, unified CPU-GPU memory space, and other memory complexities. As HPC applications get ported to run on the new supercomputers with GPUs, they will have to rely on a range of optimizations including data placement optimization to reach desired performance.

Studies [4, 8] have shown that placing data onto the proper part of a memory system also known as *data placement optimization*,

has significant impact on program performance; it is able to frequently speed up carefully written GPU kernels by over 50% (up to 13.5× on an MPEG video encoding kernel [7]). State-of-art literature explored different techniques to optimize data placement on older generations of GPUs. A portable compiler/runtime system for deciding the optimal GPU data placement is proposed in PORPLE [3, 5]. Internally it uses a lightweight performance model based on cache reuse distance to determine the data placement policy. Huang and Li [6] have proposed a new approach by analyzing correlations among different data placements. It then uses a sample placement policy to predict the performance for other placements on a Kepler GPU. A rule-based system to guide memory placement on Tesla GPUs based on data access patterns has been introduced in Jang et al. [7]. Mei et al. [11] and Jia et al. [9] have proposed microbenchmarking to analyze and expose cache parameters of Fermi, Kepler, Maxwell and Volta respectively.

The novelty in this work is in predicting optimal data placement on newer GPUs. Machine learning-based approach helps to automate the decision making process for optimal data placement without modifying the implementation of the underlying algorithm. Prior state-of-art approaches used offline profiling analysis, but were unable to capture the required features at runtime. By observing the runtime features of just the global/default version of a code, the proposed approach is able to determine which memory placement will yield best performance. In this work, we use a machine learning-based approach where a classifier model is trained once offline. During inference, runtime features are captured and passed as an input feature vector to the model that determines the optimal data placement location.

This paper makes the following novel contributions:

- determining the optimal data placement location on-the-fly during run-time
- providing a simple, lightweight solution that is applicable to diverse applications
- introducing an approach and supplying data which can be reused for other optimizations, such as determining optimal data layouts.

2 MOTIVATION

In this section, we demonstrate how the data placement in memory could impact the program execution time. Here we first provide a brief description of GPU memory hierarchy and then show how different placement of data onto various types of memories impact the kernel execution time.

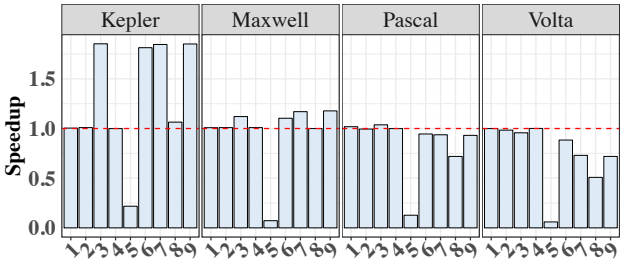
2.1 GPU Memory Structure

GPUs have a highly complex memory hierarchy in order to exploit their massive parallel computing potentials. A high-level overview of the major types of memories listed in Table 1, as exposed to programmers via the CUDA API is listed as follows:

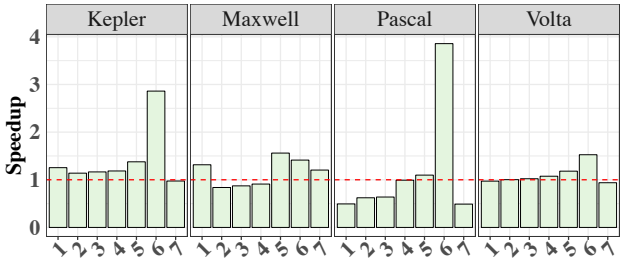
- **Global Memory**: This largest off-chip memory serves as default, main memory on a GPU. Global memory accesses are characterized with limited memory bandwidth and long latencies compared with on-chip memory or cache.
- **Shared Memory**: This on-chip memory is characterized by low-latency and high-bandwidth. It is software-managed and is accessible by active threads that belong to a streaming multiprocessor (SM) unit on a GPU.
- **Constant Memory**: This is a predefined within the global memory space and is set as read-only. The memory space is globally visible to all threads.
- **Texture Memory:** Similar to read-only constant memory, texture memory is off-chip memory space that is optimized for 2D spatial locality. It is suited for threads that access memory addresses that are closer to each other in 2D.

Memory type	Location	Access	Cached	Scope
Global	Off-Chip	Read-Write	Y	Global
Shared	On-Chip	Read-Write	N	SM
Constant	Off-Chip	Read-only	Y	Global
Texture	Off-Chip	Read-only	Y	Global

Table 1: GPU memory types



(a) Speedup of Sparse Matrix Vector Multiplication



(b) Speedup of Matrix Multiplication

Figure 2: Impact of data placement on kernel performance across different GPUs. Speedups over default vary with different memory variants shown on x-axis.

Although all NVIDIA GPUs have a similar high-level design, different generations of GPUs introduce new memory properties or implement the same memories using different physical organization. For example, Fermi GPUs introduced true cache hierarchy for global memory while previous GPUs did not have such caches. On Kepler GPUs, L1 cache and shared memory are combined together while texture cache is backed by its own memory chip. Pascal and Maxwell GPUs have dedicated memory for their shared memory. For the latest Volta GPUs, L1 cache, shared memory and texture cache are all merged into a single unified memory, All these hardware changes have a direct impact on memory optimization.

2.2 Preliminary Experiments

Preliminary experiments show that the performance achievable from different data placements can be specific to applications and platforms. From the graphs in Figure 2, it is evident that the version of the hardware and the application both heavily influence the kernel performance. These graphs show two benchmarks utilizing different GPU memories, numbered across the x-axis and their speedup compared to the global version of the benchmark. Table 2 shows what different numbered versions correspond to for the SPMV benchmark (for brevity, only one table is shown). Figure 2a shows that the performance of SPMV application, which is a sparse matrix vector application, generally worsens with the latest version of NVIDIA GPUs when utilizing non-global memory. However for MM, the matrix-matrix multiplication benchmark, performance improvement can still be seen on the latest platforms (Volta and Pascal) using non-global memory as shown in Figure 2b. Both benchmarks show that for many applications it is clearly important to get the data placement right. Applications optimized for a particular platform will not necessarily retain that performance on future platforms and indeed future platforms with advanced hardware may not require optimizing at all.

Version	Description
1	rows array in shared
2	rows array in constant
3	vector array in texture1D, rows in shared
4	matrix values in texture1D
5	vector array in constant, rows in texture1D
6	vector array in texture
7	matrix values and columns in texture1D, rows in constant
8	matrix values, columns and vector in texture1D
9	matrix values, columns, rows and vector in texture1D

Table 2: Memory configuration details of SPMV benchmark

3 APPROACH

The machine learning model for directly predicting the appropriate placement plan is based on data access patterns and memory system specifications. The workflow involves two phases, namely *offline training* and *online inference*, as shown in Figure 3. The offline training phase involves the construction of a classifier using supervised learning. It takes the feature vector of the kernel and data as inputs, and predicts the best data placement in memory. As with any machine learning approach, we investigate different classifiers such as Random Forests, Support Vector Machines, AttributedClassifiers and pick one that yields the highest prediction accuracy and confidence. To train the classifiers, we obtain the training data by evaluating representative kernels with various data placements and labeling the samples with observed best execution times.

3.1 Offline Training

3.1.1 Design of Training Experiments. The training experiments are set up to run scripts over a selection of benchmarks with varying types of data placements and data and thread block sizes across the different GPU platforms listed in Section 4. First, the best performing versions are found by measuring the program execution

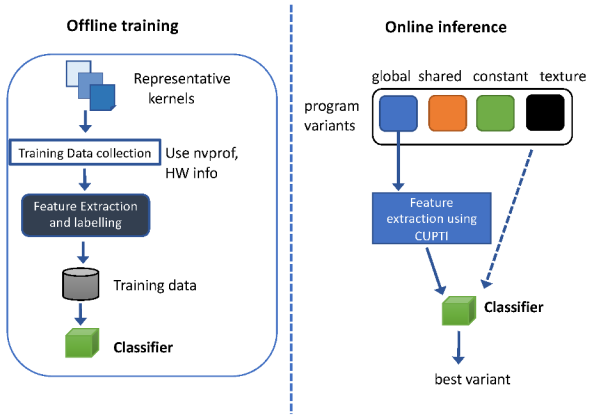


Figure 3: Workflow of the proposed machine learning-based data placement engine.

times of all possible memory variants, for a variety of data sizes and thread block sizes of each benchmark. Then the global memory versions are profiled for their feature values, selecting only those metrics and events available on all platforms. These features are paired with class labels of the best versions available for a given benchmark of a data size and thread block size. Feature values are first normalized and then re-sampled to ensure a larger spread for use. The benchmarks along with the hardware platforms can be found in Section 4.

3.1.2 Feature Engineering: As with any machine learning-based models, the classifier to make accurate predictions for the data placement, relies heavily on the input features. Here, we present how the features are extracted and important ones are selected.

(a) Feature Extraction: We use the NVIDIA profiling tool `nvprof` to extract the hardware counters and performance metrics that compose the features and use the `CUPTI` API [12] to extract these features from the codes at runtime. Hardware parameters obtained from extensive microbenchmarking [10] are then appended to each sample of features obtained `nvprof`. The raw training data set is obtained by running 8 benchmark programs with various combinations of 4 memory variants, utilizing 3 variations of data sizes and 3 different thread block sizes on 4 GPU architectures for at least ten iterations. We have 4,876 samples with 241 features per sample collected from `nvprof` and hardware. This number is larger than expected due to there being more than one version for certain benchmarks as well as additional iterations that may have been run. The benchmarks used range from 1-7 input arrays which can potentially be placed in an alternative memory.

(b) Dimensionality Reduction: The dimensionality of the features is reduced from 241 to a minimal set as shown in Table 3. These significant ones are obtained using a correlation-based feature selection (CFS) algorithm. Such reduction in the required features helps to extract only those features that are important to the model.

3.1.3 Model Training: With the reduced feature set, we evaluated different classifiers to compare respective prediction accuracies, obtained using 10-fold cross validation. Such a strategy ensures that the model does not overfit the training data and the reported

Feature	Description
achieved_occupancy	ratio of average active warps to maximum number of warps
active_warps	average of active warps per cycle
l2_subp1_write_sysmem_sector_queries	number of system memory write requests to slice 1 of L2 cache
l2_subp0_total_write_sector_queries	total number of memory write requests to slice 0 of L2 cache
l2_subp0_total_read_sector_queries	total number of memory read requests to slice 0 of L2 cache
l2_subp0_read_sector_misses	total number of memory read misses from slice 0 of L2 cache
fb_subp1_write_sectors	number of DRAM write requests to sub partition 1
dram_read_throughput	device memory read throughput
eligible_warps_per_cycle	average number of warps eligible to issue per active cycle
l2_read_transactions	memory read transactions at L2 cache for all read requests
l2_write_transactions	memory write transactions at L2 cache for all write requests
gld_throughput	global memory load throughput
flop_count_sp	single-precision FLOPS executed by non-predicated threads
flop_count_sp_special	single-precision special FLOPS executed by non-predicated threads
warp_nonpred_execution_efficiency	ratio of average active threads executing non-predicated instructions to the maximum number of threads
warp_execution_efficiency	ratio of average active threads to the maximum number of threads

Table 3: List of selected features to train the classifiers

Classifier	Prediction accuracy
RandomForest	95.7%
LogitBoost	95.5%
IterativeClassifierOptimizer	95.5%
SimpleLogistic	95.4%
JRip	95.0%

Table 4: Classifiers and their prediction accuracies

prediction accuracies are based on unseen test data. A selection of classifiers with high prediction accuracies is listed in Table 4.

3.2 Online Inference

The classifier built in the offline training phase is then used for predicting the best data placement for new applications at runtime through the `CUPTI` profiler API. Honing in on the fewest number of features needed, the application is profiled in real-time, the features are then fed as input to the classifier. Based on the prediction, the best version of the application is then executed. The assumption here is that different memory variants of the code already exist and the appropriate version that is determined by the classifier is executed at runtime.

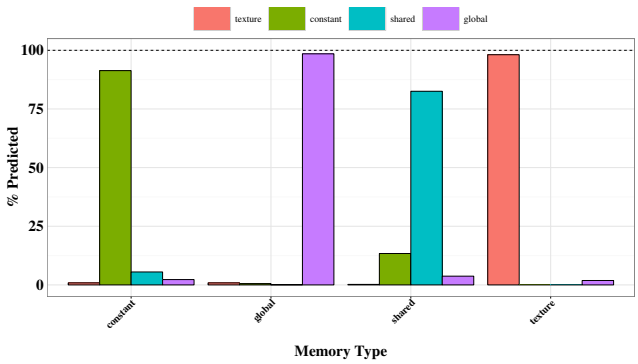


Figure 4: Graph showing the model-predicted memory classes with the best performing memory variants. Across all GPUs the model correctly predicts the best performing variant in at least 80% and up to 95% cases.

4 EVALUATION

Here we list the hardware and software platforms used to evaluate the proposed approach, followed by the experimental results that show how close the model predicted memory variant is with the best possible one. Table 5 shows the four machines used for our experiments. They contain four generations of GPUs namely Kepler, Maxwell, Pascal and Volta. The programs run include *Sparse Matrix Vector Multiplication* (SPMV), *Molecular Dynamics Simulation* (MD), *Computational Fluid Dynamics Simulation* (CFD), *Matrix-Matrix Multiplication* (MM), *ParticleFilter*, *ConvolutionSeparable*, *Stencil27*, and *Matrix Transpose*.

In the first run, the best performing version of each benchmark is determined. The benchmarks were run using a script to collect average values over ten iterations and the median of these values is taken. Each kernel was run for five times to warm up the GPU before timings were taken. In order to collect consistent performance times, the performance benchmarks used `cudaEventRecord()` to denote kernel start and stop places and `cudaEventElapsedTime()` to calculate the time elapsed. `cudaDeviceSynchronize()` was used after these calls and data transfer times are excluded in order to isolate performance differences from memory usage.

5 ANALYSIS

In this section, we demonstrate how well the `JRIP` model performs and the accuracy in its predictions for each of the benchmarks across the different platforms. We have selected this model because it is the most portable of those available and all of the top performing models hover around ~95% prediction accuracies. This model has a relative absolute error of ~9.5%. The prediction accuracies obtained from other classifiers are listed in Table 4. The results of our machine learning model experiments show that we are able to get very good results from tree-based models. Even if the model may not accurately predict the best performing data placement, it may still predict a better data placement than the global version.

The graph in Figure 4 shows the results for the best data placement class comparing the model predicted memory variants with

	Kepler (K40)	**Maxwell (M60)**	**Pascal (P100)**	**Volta (V100)**
CPU	IBM Power8 @2.2GHz	Intel Xeon E5-2670 @2.60 GHZ	IBM Power8 @2.2GHz	Intel Xeon E5-2699 @2.20GHz
Computation capability	3.5	5.2	6.0	7.0
SMs	15	16	56	80
Cores/SM	192 SP cores/64 DP cores	128 cores	64 cores	64 SP cores/32 DP cores
Texture Units/SM	16	8	4	4
Register File Size/SM	256 KB	256 KB	256 KB	256 KB
L1 Cache/SM	Combined 64K L1+Shared	Combined 24KB	Combined 24 KB	128 KB Unified
Texture Cache	48KB			
Shared Memory/SM	Combined 64K L1+Shared	96 KB	64 KB	
L2 Cache	1536 KB	2048 KB	4096 KB	6144KB
Constant Memory	64 KB	64 KB	64 KB	64 KB
Global Memory	12 GB	8 GB	16 GB	16 GB

Table 5: Key specifications of selected GPUs of different generations

the best performing ones. We found that the memory class predicted by the machine learning model is accurate in at least 80% and up to 95% of cases. Specially, given a global memory variant, this model is able to correctly predict which of the memory variants would yield the best performance (lowest execution time). These results are averaged across different GPUs.

It can be observed from the figure that, except for few instances, the offline trained model is able to correctly classify the best memory variant by observing the default global variant. The model correctly classifies global and texture memory variants to the actual best performing variants. The percentage of correct predictions is however lower for shared memory variants.

6 CONCLUSION

We have presented an automated approach to data placement optimization using machine learning to tune applications on GPUs. We built a classifier to determine the memory variant that could yield the best performance. The evaluations demonstrate that the model predictions are nearly as accurate as the best achievable cases.

This work has shown that there is an immense possibility for machine learning to be applied to automate data placement optimizations. As part of the future work, this technique would be integrated into an automated workflow and have runtime code generation of the appropriate memory variant based on the model decision. Ideally, a compiler would handle these low-level details. Additionally, the overhead of using `CUPTI` profiling tools interface means that performance suffers. Essentially, the profiler must run the kernel of interest for each metric or event, meaning that extra iterations are added. This could be alleviated by running cut down data sizes. Finally, this work could easily be applied to other areas such as different data layout optimizations.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 18-ERD-006. (LLNL-CONF-758021)

REFERENCES

[1] 2018. Sierra- next generation HPC system at LLNL. https://computation.llnl.gov/computers/sierra. (2018).
[2] 2018. Summit at Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/summit/. (2018).
[3] Guoyang Chen, Xipeng Shen, Bo Wu, and Dong Li. 2017. Optimizing data placement on GPU memory: A portable approach. *IEEE Trans. Comput.* 66, 3 (2017), 473–487.
[4] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An Extensible Optimizer for Portable Data Placement on GPU. In *The 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 88–100.
[5] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 88–100.
[6] Yingchao Huang and Dong Li. 2017. Performance modeling for optimal data placement on GPU with heterogeneous memory systems. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 166–177.
[7] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2010. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel & Distributed Systems* 1 (2010), 105–118.
[8] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *The 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 1–14.
[9] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).
[10] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018). arXiv:1804.06826 http://arxiv.org/abs/1804.06826
[11] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 72–86.
[12] NVIDIA. 2018. CUDA Profiling Tools Interface. (2018). https://developer.nvidia.com/cuda-profiling-tools-interface
[13] CUDA NVIDIA. 2007. NVIDIA CUDA programming guide (version 1.0). *NVIDIA: Santa Clara, CA* (2007).

On the Applicability of PEBS based Online Memory Access Tracking for Heterogeneous Memory Management at Scale

Aleix Roca Nonell, Balazs Gerofi‡, Leonardo Bautista-Gomez,
Dominique Martinet†, Vicenç Beltran Querol, Yutaka Ishikawa‡
Barcelona Supercomputing Center, Spain
†CEA, France
‡RIKEN Center for Computational Science, Japan
{aleix.rocanonell,leonardo.bautista,vbeltran}@bsc.es,dominique.martinet@cea.fr,{bgerofi,yutaka.ishikawa}@riken.jp

ABSTRACT

Operating systems have historically had to manage only a single type of memory device. The imminent availability of heterogeneous memory devices based on emerging memory technologies confronts the classic single memory model and opens a new spectrum of possibilities for memory management. Transparent data movement between different memory devices based on access patterns of applications is a desired feature to make optimal use of such devices and to hide the complexity of memory management to the end user. However, capturing memory access patterns of an application at runtime comes at a cost, which is particularly challenging for large-scale parallel applications that may be sensitive to system noise.

In this work, we focus on the access pattern profiling phase prior to the actual memory relocation. We study the feasibility of using Intel's Processor Event-Based Sampling (PEBS) feature to record memory accesses by sampling at runtime and study the overhead at scale. We have implemented a custom PEBS driver in the IHK/-McKernel lightweight multi-kernel operating system, one of whose advantages is minimal system interference due to the lightweight kernel's simple design compared to other OS kernels such as Linux. We present the PEBS overhead of a set of scientific applications and show the access patterns identified in noise sensitive HPC applications. Our results show that clear access patterns can be captured with a 10% overhead in the worst-case and 1% in the best case when running on up to 128k CPU cores (2,048 Intel Xeon Phi Knights Landing nodes). We conclude that online memory access profiling using PEBS at large-scale is promising for memory management in heterogeneous memory environments.

CCS CONCEPTS

• **Software and its engineering → Operating systems**;

KEYWORDS

high-performance computing, operating systems, heterogeneous memory

ACM Reference Format:
Aleix Roca Nonell, Balazs Gerofi‡, Leonardo Bautista-Gomez, Dominique Martinet†, Vicenç Beltran Querol, Yutaka Ishikawa‡. 2018. On the Applicability of PEBS based Online Memory Access Tracking for Heterogeneous Memory Management at Scale. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3286475.3286477

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286477

1 INTRODUCTION

The past decade has brought an explosion of new memory technologies. Various high-bandwidth memory types, e.g., 3D stacked DRAM (HBM), GDDR and multi-channel DRAM (MCDRAM) as well as byte addressable non-volatile storage class memories (SCM), e.g., phase-change memory (PCM), resistive RAM (ReRAM) and the recent 3D XPoint, are already in production or expected to become available in the near future.

Management of such heterogeneous memory types is a major challenge for application developers, not only in terms of placing data structures into the most suitable memory but also to adaptively move content as application characteristics changes in time. Operating system and/or runtime level solutions that optimize memory allocations and data movement by transparently mapping application behavior to the underlying hardware are thus highly desired.

One of the basic requirements of a system level solution is the ability to track the application's memory access patterns in real-time with low overhead. However, existing solutions for access pattern tracking are often based on dynamic instrumentation, which have prohibitive overhead for an online approach [16]. Consequently, system level techniques targeting heterogeneous memory management typically rely on a two-phase model, where the application is profiled first, based on which the suggested allocation policy is then determined [5, 19].

Intel's Processor Event-Based Sampling (PEBS) [3] is an extension to hardware performance counters that enables sampling the internal execution state of the CPU (including the most recent virtual address accessed) and periodically storing a snapshot of it into main memory. The overhead of PEBS has been the focus of previous works [1, 15], however, not in the context of large-scale high-performance computing (HPC).

The hardware PEBS support provides a number of configuration knobs that control how often PEBS records are stored and how often the CPU is interrupted for additional background data processing. Because such disruption typically degrades performance at scale [6, 12], it is important to characterize and understand this overhead to assess PEBS' applicability for heterogeneous memory management in large-scale HPC. Indeed, none of the previous studies focusing on PEBS' overhead we are aware of have addressed large-scale environments.

We have implemented a custom PEBS driver in the IHK/McKernel lightweight multi-kernel operating system [8, 9]. Our motivation for a lightweight kernel (LWK) is threefold. First, lightweight kernels are known to be highly noise-free and thus they provide an excellent environment for characterizing PEBS' overhead. Second, McKernel has a relatively simple code-base that enables us to rapidly prototype kernel level features for heterogeneous memory management and allow direct integration with our PEBS driver. Our custom driver can be easily configured and enables fine-grained tuning of parameters that are otherwise not available in the Linux driver (see Section 3 for more details). Finally, the Linux PEBS driver on the platform we used in this study, i.e., the Oakforest-PACS machine [13] based on Intel's Xeon Phi Knight's Landing chip, was not available.

As the baseline for OS level hierarchy memory management, we aimed at answering the following questions. What is the overhead of real-time memory accesses tracking at scale? What is the trade-off between sampling granularity and the introduced overhead? Is it feasible to rely on PEBS for collecting such information online?

Specifically, in this paper we make the following contributions:

- An implementation of a custom PEBS driver in an LWK with the ability of fine-tuning its parameters
- Systematic evaluation of PEBS' overhead on a number of real HPC applications running at large scale
- Demonstration of captured memory access patterns as the function of different PEBS parameters

Previous studies have reported PEBS failing to provide increased accuracy with reset values (see Section 2.1) lower than 1024 [1, 15] as well as the Linux kernel becoming unstable when performing PEBS based sampling on high frequency [18]. On up to 128k CPU cores (2,048 Xeon Phi KNL nodes), we find that our custom driver captures increasingly accurate access patterns reliably even with very low reset values. Across all of our workloads, PEBS incurs an overhead of 2.3% on average with approximately 10% and 1% in the worst and best cases, respectively.

The rest of this paper is organized as follows. We begin by explaining the background and motivations in Section 2. We describe the design and implementation of our custom PEBS driver in Section 3. Our large-scale evaluation is provided in Section 4. Section 5 discusses related work, and finally, Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

This section lays the groundwork for the proposed driver architecture by providing background information on Intel's Processor Event-Based Sampling facility [3] and the IHK/McKernel lightweight multi-kernel OS [7–9].

2.1 Processor Event-Based Sampling

Processor Event-Based Sampling (PEBS) is a feature of some Intel microarchitectures that builds on top of Intel's Performance Counter Monitor (PCM).

The PCM facility allows to monitor a number of predefined processor performance parameters (hereinafter called "events") by counting the number of occurrences of the specified events[1] in a set of dedicated hardware registers. When a PCM counter overflows an interrupt is triggered, which eases the process of sampling.

PEBS extends the idea of PCM by transparently storing additional processor information while monitoring a PCM event. However, only a small subset of the PCM events actually support PEBS. A "PEBS record" is stored by the CPU in a user-defined memory buffer when a configurable number of PCM events, named "PEBS reset counter value" or simply "reset", occur. The actual PEBS record format is microarchitecture dependent, but it generally includes the set of general-purpose registers.

A "PEBS assist" in Intel nomenclature is the action of storing the PEBS record into the CPU buffer. When the record written in the last PEBS assist reaches a configurable threshold inside the CPU PEBS buffer, an interrupt is triggered. The interrupt handler should process the PEBS data and clear the buffer, allowing the CPU to continue storing more records. The PCM's overflow interrupt remains inactive while a PCM event is being used with PEBS.

2.2 Lightweight Multi-kernels

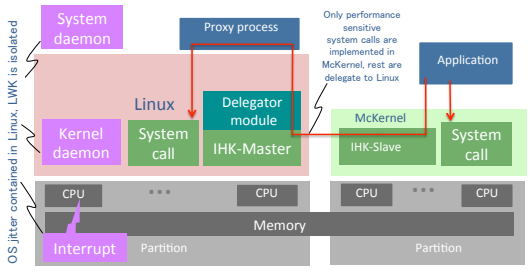


Figure 1: Overview of the IHK/McKernel architecture.

Lightweight multi-kernels emerged recently as a new operating system architecture for HPC, where the basic idea is to run Linux and a LWK side-by-side in compute nodes to attain the scalability properties of LWKs and full compatibility with Linux at the same time. IHK/McKernel is a multi-kernel OS developed at RIKEN, whose architecture is depicted in Figure 1. A low-level software infrastructure, called Interface for Heterogeneous Kernels (IHK) [21], provides capabilities for partitioning resources in a many-core environment (e.g., CPU cores and physical memory) and it enables management of lightweight kernels. IHK is capable of allocating and releasing host resources dynamically and no reboot of the host machine is required when altering its configuration. The latest version of IHK is implemented as a collection of Linux kernel modules

[1]The exact availability of events depends on the processor's microarchitecture. However, a small set of "architectural performance events" remain consistent starting from the Intel Core Solo and Intel Core Duo generation.

without any modifications to the Linux kernel itself, which enables relatively straightforward deployment of the multi-kernel stack on a wide range of Linux distributions. Besides resource and LWK management, IHK also facilitates an Inter-kernel Communication (IKC) layer.

McKernel is a lightweight co-kernel developed on top of IHK. It is designed explicitly for HPC workloads, but it retains a Linux compatible application binary interface (ABI) so that it can execute unmodified Linux binaries. There is no need for recompiling applications or for any McKernel specific libraries. McKernel implements only a small set of performance sensitive system calls and the rest of the OS services are delegated to Linux. Specifically, McKernel provides its own memory management, it supports processes and multi-threading, it has a simple round-robin co-operative (tick-less) scheduler, and it implements standard POSIX signaling. It also implements inter-process memory mappings and it offers interfaces for accessing hardware performance counters.

For more information on system call offloading, refer to [8], a detailed description of the device driver support is provided in [9]. Recently we have demonstrated that lightweight multi-kernels can indeed outperform Linux on various HPC mini-applications when evaluated on up to 2,048 Intel Xeon Phi nodes interconnected by Intel's OmniPath network [7]. As mentioned earlier, with respect to this study, one of the major advantages of a multi-kernel LWK is the lightweight kernel's simple codebase that enables us to easily prototype new kernel level features.

3 DESIGN AND IMPLEMENTATION

This section describes the design and implementation of the McKernel PEBS driver. Figure 2 shows a summary of the entire PEBS records lifecycle.

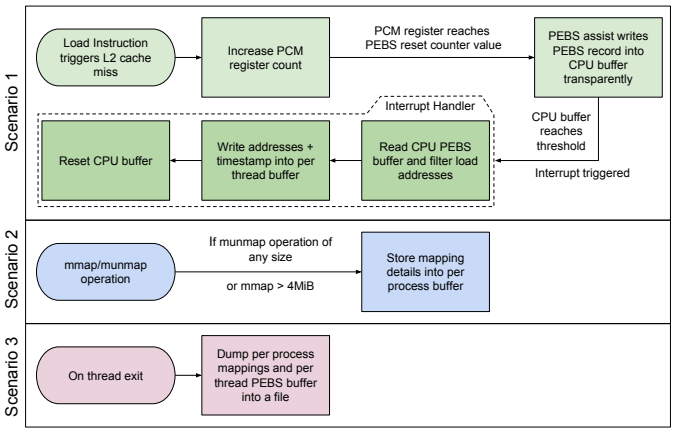


Figure 2: Memory addresses acquisition processes using Intel's PEBS facility in IHK/McKernel

McKernel uses PEBS as a vehicle to keep track of memory addresses issued by each monitored system thread. Ideally, McKernel would keep track of all load and store instructions. However, this is not supported by all Intel microarchitectures. In particular, our test environment powered by the Intel Knights Landing processor only supports recording the address of load instructions that triggered some particular event. PEBS records are always associated with a PCM event. The most general KNL PCM events that support load address recording are L2_HIT_LOADS and L2_MISS_LOADS which account for L2 hits and L2 misses, respectively.

Both the count of L2 misses and L2 hits in a page boundary for a given time frame can be used as a metric that determines how likely is the page to be accessed in the future. A page with a high count of either L2 misses or L2 hits reveals that the page is under memory pressure. In the case of misses, we additionally know that the cache is not able to hold the pages long enough to be reused. And in the case of hits, we know that either pages are accessed with high enough frequency to remain in the cache or simply the whole critical memory range fits into the cache.

In principle, a page with a high L2 miss ratio seems to be a good candidate for being moved into a faster memory device because missing the L2 in the case of KNL means that data must be serviced from either main memory or the L2 of another core. However, the same page might actually have a higher ratio of L2 hits, indicating that another page with a lower hit ratio might benefit still more from being moved. In consequence, fair judgment should take into consideration both events. Unfortunately, KNL features a single PCM counter with PEBS support, which means that sampling both events requires to perform dynamic switching at runtime. Nonetheless, the purpose of this work is just a step behind. Our objective is to focus on the study of a single PEBS enabled PCM counter at scale. Therefore, for simplicity, we decided to rely on the L2_MISS_LOADS event to record the load addresses.

McKernel initializes the PEBS CPU data structures at boot time on each CPU. Processes running in McKernel will enable PEBS on all the CPUs where its threads are running as soon as they start. As long as the threads are being run, PEBS assists will write PEBS records into the CPU's buffer transparently regardless of their execution context (user or kernel space).

The PEBS record format for the Knights Landing architecture consists of (among others) the set of general-purpose registers and the address of the load instruction causing the record dump (PEBS assist) if applicable. In total, 24 64-bit fields are stored, adding up to a total of 192 bytes for each PEBS record. There is no timestamp information stored in each PEBS record so it is not possible to know exactly when the record took place.

When the PEBS remaining capacity reaches the configured threshold, an interrupt is triggered. The PEBS interrupt handler filters all fields in the PEBS records but the load address and saves them into a per-thread circular buffer. Then, the CPU PEBS buffer is reset, allowing the CPU to continue storing records. Altogether with the load addresses, a timestamp is saved at the time the interrupt handler is running. This timestamp tags all the PEBS records processed in this interrupt handler execution for posterior analysis.

When each of the application's threads exit, the entire contents of the per-thread buffer is dumped into a file. We have developed a small python visualization tool to read and generate plots based on the information provided.

The registered load addresses might not belong to application-specific user buffers but from anywhere in the address space. For offline visualization purposes we are mostly interested in profiling the application's buffers and hence, it is convenient to provide some means to filter the undesired addresses. Load addresses can

be sparse, and visualizing the entire address space of an application to detect patterns might be difficult. It is important to notice that filtering is not a requirement for online monitoring of high demanded pages, this is only necessary for visualization.

A simple heuristic to do so is to filter out all addresses of small mappings. To minimize the impact of filtering, the postprocessing is done offline in our visualization script. Hence, McKernel only keeps track of all mappings greater than four megabytes by storing its start addresses, the length and the timestamp at which the operation completed. All munmap operations are also registered regardless of its size because they might split a bigger tracked area. The mappings information are stored into a per-process buffer, shared by all threads using a lock-free queue. The per-process mappings buffer is also dumped into the PEBS file at each thread's termination time.

Our PEBS addresses viewer loads the file and reconstructs the processes virtual memory mappings history based on the mmap and munmap memory ranges and timestamps. Then, it reads all the registered PEBS load addresses and classifies them into the right spatial and temporal mapping or discards them if no suitable mapping is found. Finally, individual plots are shown per mapping.

The PEBS data acquisition rate is controlled by the configurable number of events that trigger a PEBS assist and the size of the CPU PEBS buffer (which indirectly controls the threshold that triggers an interrupt). We have added a simple interface into McKernel to dynamically configure these parameters at application launch time by resizing the CPU buffer and reconfiguring the PEBS MSR registers as requested. This differs from the current Linux Kernel driver in which it is only possible to configure the reset counter value but not the PEBS buffer size.

It would be ideal to have a big enough CPU buffer to hold all load addresses the application generates to both reduce the memory movements between buffers and to suppress the interrupts overhead. However, having a small interrupt rate also diffuses the time perception of memory accesses because timestamps are associated with PEBS records in the interrupt handler. Therefore, this implementation actually requires to set up a proper interrupt rate to understand the evolution of memory accesses in time. Note that instead of relying on the interrupt handler to harvest the PEBS CPU buffer, another option is to dedicate a hardware thread to this task. We plan to implement this option in the near future.

4 EVALUATION

4.1 Experimental Environment

All of our experiments were performed on Oakforest-PACS (OFP), a Fujitsu built, 25 petaflops supercomputer installed at JCAHPC, managed by The University of Tsukuba and The University of Tokyo [13]. OFP is comprised of eight-thousand compute nodes that are interconnected by Intel's Omni Path network. Each node is equipped with an Intel® Xeon Phi™ 7250 Knights Landing (KNL) processor, which consists of 68 CPU cores, accommodating 4 hardware threads per core. The processor provides 16 GB of integrated, high-bandwidth MCDRAM and it also is accompanied by 96 GB of DDR4 RAM. The KNL processor was configured in Quadrant flat mode; i.e., MCDRAM and DDR4 RAM are addressable at different physical memory locations and are presented as separate NUMA nodes to the operating system.

The software environment was as follows. Compute nodes run CentOS 7.4.1708 with Linux kernel version `3.10.0-693.11.6`. This CentOS distribution contains a number of Intel supplied kernel level improvements specifically targeting the KNL processor that were originally distributed in Intel's XPPSL package. We used Intel MPI Version 2018 Update 1 Build 20171011 (id: 17941) in this study.

For all experiments, we dedicated 64 CPU cores to the applications (i.e., to McKernel) and reserved 4 CPU cores for Linux activities. This is a common scenario for OFP users where daemons and other system services run on the first four cores even in Linux only configuration.

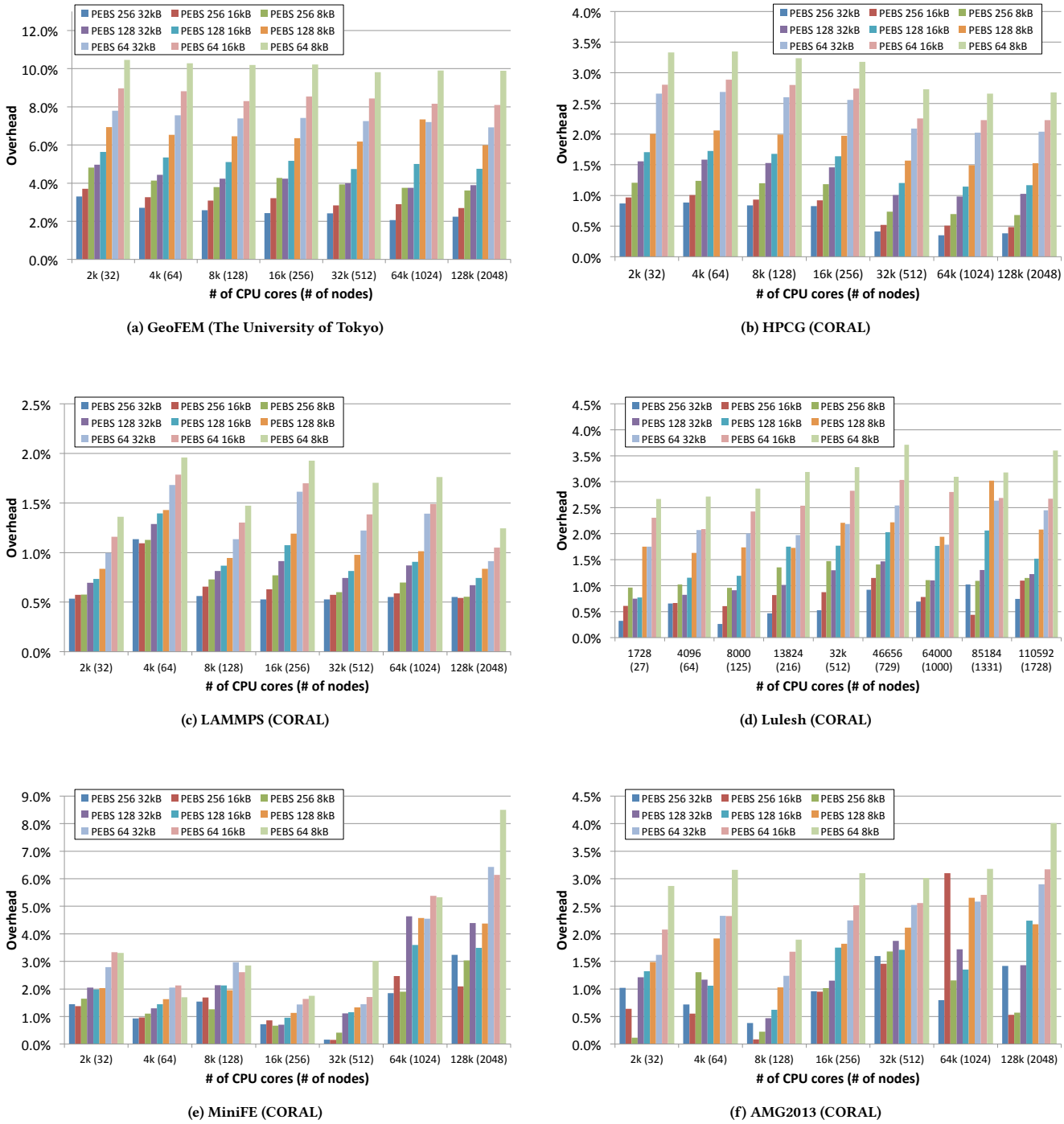
4.2 Mini-applications

We used a number of mini-applications from the CORAL benchmark suite [2] and one developed at the The University of Tokyo. Along with a brief description, we also provide information regarding their runtime configuration.

- **GeoFEM** solves 3D linear elasticity problems in simple cube geometries by parallel finite-element method [17]. We used weak-scaling for GeoFEM and ran 16 MPI ranks per node, where each rank contained 8 OpenMP threads.
- **HPCG** is the High Performance Conjugate Gradients, which is a stand-alone code that measures the performance of basic operations in a unified code for sparse matrix-vector multiplication, vector updates, and global dot products [4]. We used weak-scaling for HPCG and ran 8 MPI ranks per node, where each rank contained 8 OpenMP threads.
- **Lammps** is a classical molecular dynamics code, an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator [20]. We used weak-scaling for Lammps and ran 32 MPI ranks per node, where each rank contained four OpenMP threads.
- **miniFE** is a proxy application for unstructured implicit finite element codes [11]. We used strong-scaling for miniFE and ran 16 MPI ranks per node, where each rank contained four OpenMP threads.
- **Lulesh** is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics code which was originally defined and as one of five challenge problems in the DARPA UHPC program [14]. We used weak-scaling for Lulesh and ran 8 MPI ranks per node, where each rank contained 16 OpenMP threads.
- **AMG2013** is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids [10]. We used weak-scaling for AMG and ran 16 MPI ranks per node, where each rank contained 16 OpenMP threads.

4.3 Results

For each workload described above, we use nine different PEBS configurations. We scale the PEBS reset value from 256, through 128 to 64 and used PEBS per-CPU buffer sizes of 8kB, 16kB and 32kB. As mentioned earlier, the reset value controls the sampling granularity while the PEBS buffer size impacts the PEBS IRQ frequency. We emphasize again that contrary to previous reports on PEBS' inability to provide increased accuracy with reset values lower than 1024 [1, 15, 18], we find very clear indications that obtaining increasingly



(a) GeoFEM (The University of Tokyo)

(b) HPCG (CORAL)

(c) LAMMPS (CORAL)

(d) Lulesh (CORAL)

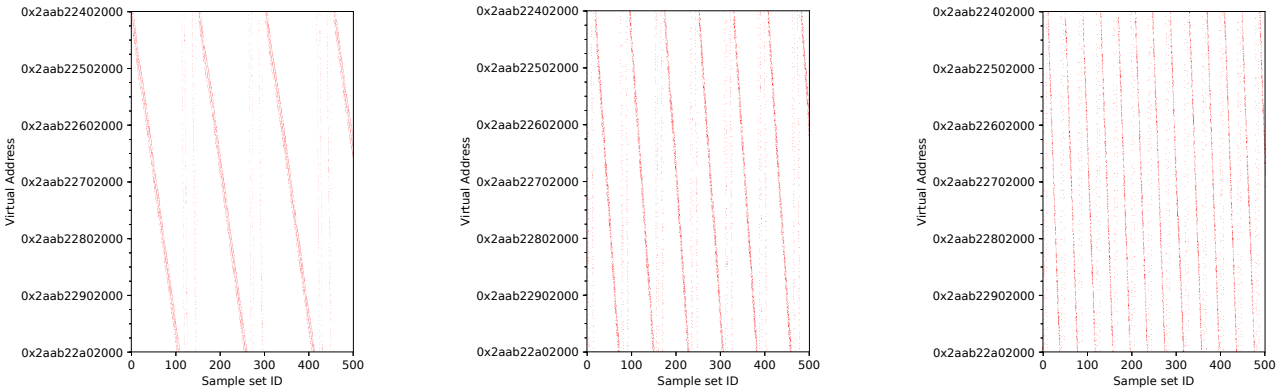
(e) MiniFE (CORAL)

(f) AMG2013 (CORAL)

Figure 3: PEBS overhead for GeoFEM, HPCG, LAMMPS, Lulesh, MiniFE and AMG on up to 2,048 Xeon Phi KNL nodes

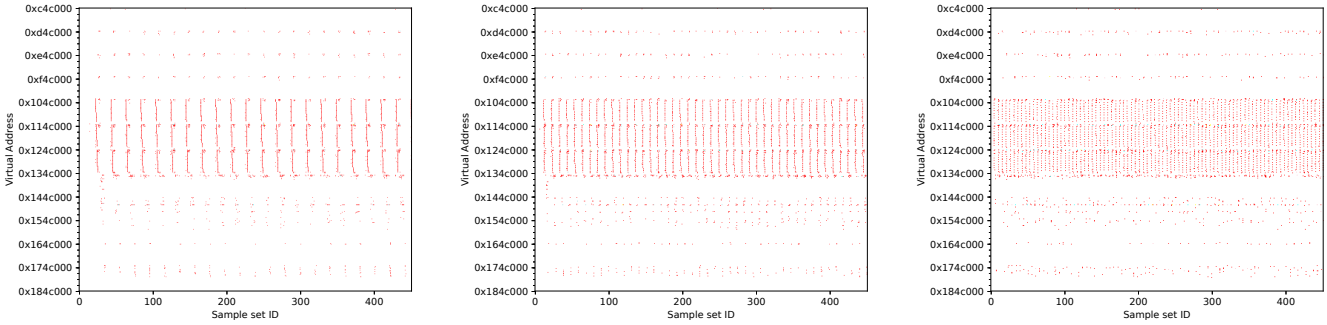
accurate samples with lower reset values is possible, for which we provide more information below.

We ran each workload for all configurations scaling from 2,048 to 128k CPU cores, i.e., from 32 to 2,048 compute nodes, respectively. We compare individually the execution time of each benchmark run on McKernel with and without memory accesses tracking enabled. We report the average value of three executions, except for a few long-running experiments, where we took only two samples (e.g.,



(a) PEBS reset = 64 **(b) PEBS reset = 128** **(c) PEBS reset = 256**

Figure 4: MiniFE access pattern with different PEBS reset values (8kB PEBS buffer)



(a) PEBS reset = 64 **(b) PEBS reset = 128** **(c) PEBS reset = 256**

Figure 5: Lulesh access pattern with different PEBS reset values (8kB PEBS buffer)

for GeoFEM). Note that all measurements were taken on McKernel and no Linux numbers are provided. For a detailed comparison between Linux and McKernel, refer to [7].

Figure 3 summarizes our application level findings. The X-axis represents node counts while the Y-axis shows relative overhead compared to the baseline performance. For each bar in the plot, the legend indicates the PEBS reset value and the PEBS buffer size used in the given experiment. The general tendency of overhead for most of the measurements matched our expectations, i.e., the most influential factor in performance overhead is the PEBS reset value, whose impact can be relaxed to some extent by adjusting the PEBS buffer size.

Across all workloads, we observe the largest overhead on GeoFEM (shown in Figure 3a) when running with the lowest PEBS reset value of 64 and the smallest PEBS buffer of 8kB, where the overhead peaked at 10.2%. Nevertheless, even for GeoFEM a less aggressive PEBS configuration, e.g., a reset value of 256 with 32kB PEBS buffer size induces only up to 4% overhead.

To much of our surprise, on most workloads PEBS's periodic interruption of the application does not imply additional overhead as we scale out with the number of compute nodes. In fact, on some of the workloads, e.g., HPCG (shown in Figure 3b) and Lammps (shown in Figure 3c) we even observe a slight decrease in overhead for which we have currently no precise explanation and for which identifying its root cause further investigation is required. Note that both of these workloads were weak scaled and thus are presumed to compute on a quasi-constant amount of per-process data irrespective of scale.

One particular application that did experience growing overhead as the scale increased is MiniFE, shown in Figure 3e. MiniFE was the only workload we ran in strong-scaled configuration and our previous experience with MiniFE indicates that it is indeed sensitive to system noise [7]. Despite the expectation that due to the decreasing amount of per-process data at larger node counts the PEBS' overhead would gradually diminish, the disruption from constant PEBS interrupts appears to amplify its negative impact.

To demonstrate the impact of PEBS' reset value on the accuracy of memory access tracking we provide excerpts on memory access patterns using different reset values. We have been able to observe similar memory access patterns for all benchmarks tested, but we present the results for MiniFE and Lulesh as an example. Figure 4 and Figure 5 show the heatmaps of the access patterns

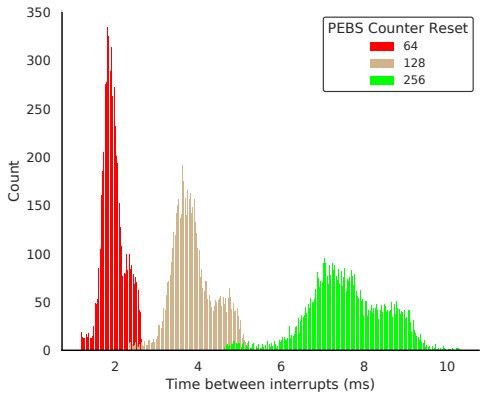


Figure 6: Distribution of elapsed time between PEBS interrupts for MiniFE with three different reset values

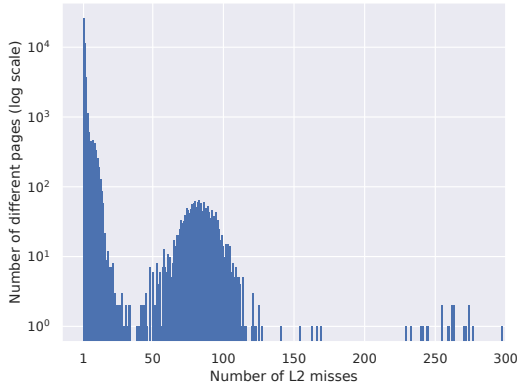


Figure 7: Access histogram per page for MiniFE execution

captured on 32 nodes for three reset values, 64, 128 and 256. The X-axis represents the sample set ID, i.e., periods of time between PEBS interrupts, while the Y-axis indicates the virtual address of the corresponding memory pages. Although PEBS addresses are captured at byte granularity, page size is the minimum unit the OS' memory manager works with. In fact, for better visibility, we show the heatmap with higher unit sizes, i.e., in blocks of 4 pages.

One of the key observations here is the increasingly detailed view of the captured access pattern as we decrease the PEBS reset counter. As seen, halving the reset value from 128 to 64 gives a 2X higher granularity per sample set, e.g., the stride access of MiniFE is stretched with respect to the sample IDs. Note that one iteration of MiniFE's buffer presented in the plot corresponds to approximately 330ms. To put the accuracy into a more quantitative from the 1536 pages of the buffer shown in the figure, PEBS with 64 reset value reports 1430 pages touched, while using reset values of 128 and 256 report 1157 and 843, respectively. To the contrary, Lulesh's plots indicate that access patterns that do not significantly change in time can be captured also with lower granularity and thus the reset value should be adjusted dynamically based on the application. Note that the number of computational nodes used affects the amount of memory each node works with and might alter the visible pattern. However, as long as the memory share per core does not fit in the L2 the patterns will generally remain similar.

The implicit effect of altering the PEBS reset counter is the increase or decrease rate of the PEBS interrupt frequency, assuming a constant workload. The capacity of controlling the interrupt rate should have a clear impact on the expected overhead, at least in noise sensitive applications such as minife. We have presented the relationship between overhead and PEBS reset counter in Figure 3 and we now show the relationship between PEBS reset counter and interrupt frequency in Figure 6. The elapsed time between interrupts is shown for three executions of MiniFE with 64, 128 and 256 values. As expected, we can see a clear correlation between the average duration and the reset counter value being the former smaller when the later decreases. We also note that the duration of the interrupt handler itself took approximately 20 thousand cycles. It is also interesting to observe the formation of two close peaks per execution. This tendency identifies two different access patterns within the application that lead to a different L2 miss generation scheme.

The presence of particularly hot pages can be easily localized by inspecting the histogram of aggregated L2 misses shown in Figure 7. The plot shows the number of different pages that had N number of L2 misses on the Y-axis, where N is shown on the X-axis. We can easily see that most of the pages in MiniFE had a small number of misses at the leftmost side of the histogram. However, the plot reveals an important group of pages above the 50 L2 misses that could be tagged as movable targets.

In summary, we believe that our large-scale results well demonstrate PEBS' modest overhead to online memory access tracking and we think that a PEBS based approach to heterogeneous memory management is worth pursuing.

5 RELATED WORK

This section discusses related studies in the domains of heterogeneous memory management and memory access tracking.

Available tools that help to determine efficient data placement in heterogeneous memory systems typically require developers to run a profile phase of their application and modify their code accordingly. Dulloor et al. proposed techniques to identify data objects to be placed into DRAM in a hybrid DRAM/NVM configuration [5]. Peng et al. considered the same problem in the context of MCDRAM/DRAM using the Intel Xeon Phi processor [19]. In order to track memory accesses, these tools often rely on dynamic instrumentation (such as PIN [16]), which imposes significant performance overhead that makes it impractical for online access tracking.

Larysch developed a PEBS system to assess memory bandwidth utilization of applications and reported low overheads, but the authors did not provide a quantitative characterization of using PEBS for this purpose [15]. Akiyama et al. evaluated PEBS overhead on a set of enterprise computing workloads with the aim of finding performance anomalies in high-throughput applications (e.g., Spark, RDBMS) [1]. PEBS has been also utilized to determine data placement in emulated non-volatile memory based heterogeneous

systems [22]. None of these works, however, have focused on exclusively studying PEBS overhead on large-scale configurations. To the contrary, we explicitly target large-scale HPC workloads to assess the scalability impacts of PEBS based memory access tracking.

Olson et al. reported in a very recent study that decreasing the PEBS reset value below 128 on Linux caused the system to crash [18]. While they disclosed results only for a single node setup, we demonstrated that our custom PEBS driver in McKernel performs reliably and induces low overheads even when using small PEBS reset values in a large-scale deployment.

6 CONCLUSION AND FUTURE WORK

This paper has presented the design, implementation and evaluation of a PEBS driver for the IHK/McKernel which aims to provide the groundwork for an OS level heterogeneous memory manager. We have shown the captured access patterns of two scientific applications and demonstrated the evolution of their resolution as we change the PEBS profiling parameters. We have analyzed the overhead impact associated with the different recording resolutions in both timing and interrupt domains at scale up to 128k CPUs (or 2,048 computer nodes) for six scientific applications. We observed overheads highly dependent on both the application behavior and the recording parameters which range between 1% and 10.2%. However, we have been able to substantially reduce the overhead of our worst-case scenario from 10.2% to 4% by adjusting the recording parameters while still achieving clearly visible access patterns. Our experience contrast with the current Linux kernel PEBS implementation which is not capable of achieving very fine-grained sample rates. We conclude that PEBS efficiency matches the basic requirements to be feasible for heterogeneous memory management but further work is necessary to quantify the additional overhead associated with using the recorded data at runtime.

Our immediate future work is to address the challenge of properly using the recorded addresses at runtime to reorganize memory pages on memory devices based on access patterns. We will study the benefits of dedicating a hardware thread to periodically harvest the CPU PEBS buffer instead of relying on interrupts that constantly pause the execution of the user processes. We also intend to deeply analyze the difference between the IHK/McKernel PEBS driver and the Linux kernel driver to better quantify the observed limitations.

ACKNOWLEDGMENT

This work has been partially funded by MEXT's program for the Development and Improvement of Next Generation Ultra High-Speed Computer Systems under its subsidies for operating the Specific Advanced Large Research Facilities in Japan. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 708566 (DURO) and agreement No 754304 (DEEP-EST).

REFERENCES

[1] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017 (ROSS '17)*. ACM, New York, NY, USA, Article 3, 8 pages.

[2] CORAL. 2013. Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/. (Nov. 2013).

[3] Intel Corporation. 2018. Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/articles/intel-sdm. (2018).

[4] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. 2015. *HPCG Benchmark: A New Metric for Ranking High Performance Computing Systems.* Technical Report UT-EECS-15-736. University of Tennessee, Electrical Engineering and Computer Science Department.

[5] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 15, 16 pages. http://doi.acm.org/10.1145/2901318.2901344

[6] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. 2008. Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 19, 12 pages.

[7] Balazs Gerofi, Rolf Riesen, Masamichi Takagi, Taisuke Boku, Yutaka Ishikawa, and Robert W. Wisniewski. 2018 (to appear). Performance and Scalability of Lightweight Multi-Kernel based Operating Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[8] Balazs Gerofi, Akio Shimada, Atsushi Hori, and Yutaka Ishikawa. 2013. Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures. In *13th Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid)*.

[9] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa. 2016. On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1041–1050.

[10] V. E. Henson and U. M. Yang. 2002. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. https://codesign.llnl.gov/amg2013.php. *Appl. Num. Math.* 41 (2002), 155–177.

[11] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. *Improving Performance via Mini-applications.* Technical Report SAND2009-5574. Sandia National Laboratories.

[12] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA. https://doi.org/10.1109/SC.2010.12

[13] Joint Center for Advanced HPC (JCAHPC). 2017. Basic Specification of Oakforest-PACS. http://jcahpc.jp/files/OFP-basic.pdf. (March 2017).

[14] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. Lawrence Livermore National Laboratory. 1–9 pages.

[15] Florian Larysch. 2016. *Fine-Grained Estimation of Memory Bandwidth Utilization.* Master Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200.

[17] Kengo Nakajima. 2003. Parallel Iterative Solvers of GeoFEM with Selective Blocking Preconditioning for Nonlinear Contact Problems on the Earth Simulator. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC)*. ACM, New York, NY, USA. https://doi.org/10.1145/1048935.1050164

[18] Matthew Benjamin Olson, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, M. Graham Lopez, and Oscar Hernandez. 2018. MemBrain: Automated Application Guidance for Hybrid Memory Systems. In *IEEE International Conference on Networking, Architecture, and Storage (NAS' 18)*. (to appear).

[19] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. RTHMS: A Tool for Data Placement on Hybrid Memory System. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, New York, NY, USA, 82–91.

[20] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-range Molecular Dynamics. (March 1995), 19 pages. https://doi.org/10.1006/jcph.1995.1039

[21] Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori, and Yutaka Ishikawa. 2014. Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures. In *21th Intl. Conference on High Performance Computing (HiPC)*.

[22] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime Data Managementon Non-volatile Memory-based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 58, 14 pages.

Exploring Allocation Policies in Disaggregated Non-Volatile Memories

Vamsee Reddy Kommareddy
University of Central Florida
Orlando, Florida, USA
vamseereddy8@knights.ucf.edu

Amro Awad
University of Central Florida
Orlando, Florida, USA
amro.awad@ucf.edu

Clayton Hughes
Sandia National Labs
New Mexico, USA
chughes@sandia.gov

Simon David Hammond
Sandia National Labs
New Mexico, USA
sdhammo@sandia.gov

ABSTRACT

Many modern applications have memory footprints that are increasingly large, driving system memory capacities higher and higher. However, due to the diversity of applications that run on High-Performance Computing (HPC) systems, the memory utilization can fluctuate widely from one application to another, which results in underutilization issues when there are many jobs with small memory footprints. Since memory chips are collocated with the compute nodes, this necessitates the need for message passing APIs to be able to share information between nodes.

To address some of these issues, vendors are exploring disaggregated memory-centric systems. In this type of organization, there are discrete nodes, reserved solely for memory, which are shared across many compute nodes. Due to their capacity, low-power, and non-volatility, Non-Volatile Memories (NVMs) are ideal candidates for these memory nodes. Moreover, larger memory capacities open the door to different programming models (more shared memory style approaches) which are now being added to the C++ and Fortran language specifications. This paper proposes a simulation model for studying disaggregated memory architectures using a publicly available simulator, SST Simulator, and investigates various memory allocation policies.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**;

KEYWORDS

Disaggregated memory system, non-volatile memory, memory-centric computing

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286480

ACM Reference Format:
Vamsee Reddy Kommareddy, Amro Awad, Clayton Hughes, and Simon David Hammond. 2018. Exploring Allocation Policies in Disaggregated Non-Volatile Memories. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3286475.3286480

1 INTRODUCTION

With the arrival of the big data era, the need for fast processing and access to shared memory structures have never been as crucial as they are today. A wide range of applications, such as high-performance database applications, graph analytics and big data applications, frequently share data between nodes. Communication between compute typically requires expensive system calls and the invocation of message passing interfaces, e.g., OpenMPI. Unfortunately, given the potentially huge amount of data shared between computing nodes, explicit communication between nodes can burden scalability and efficiency. Additionally, a recent study shows that about 80% of the jobs on HPC systems overestimate their memory requirements [2]; thus, HPC systems underutilize memory slots by dedicating them to specific jobs. Similarly, HPC architects tend to design the memory capacity per-node based on the most memory demanding applications, which also leads to underutilizing memory slots on computing nodes. In both cases, the memory consumes idle power (refresh power for DRAM) even if not used.

In addition to the underutilization and sharing overheads of the systems that couple memory with computing nodes, upgrading memory can be challenging. To take advantage of the fast-evolving memories and adapt to the new requirements of applications, system memory should have the ability to be flexibly augmented with evolving memory technologies. In systems that deploy petabytes of storage, it is important to be able to flexibly extend the data stores and ensure their robustness.

To mitigate the scalability challenges of coupled memory systems, a new design direction is evolving as a result of memory-driven applications, *disaggregated memory systems* [12, 23]. As shown in Figure 1, disaggregated memory decouples memory from computing nodes (FPGAs, GPUs or SoCs), e.g., The Machine project by HP Labs [13]. In such

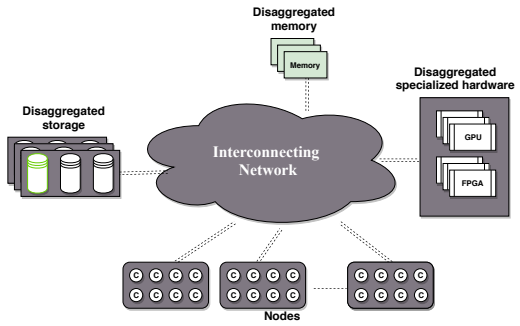


Figure 1: Disaggregated Memory System

systems, the applications can use traditional shared memory interfaces to operate on shared data by utilizing the large shared memory space. Most importantly, the shared memory space can be accessed by traditional load/store operations instead of explicitly communicating between computing nodes. Moreover, each node can request as much memory as it needs from the shared space while the rest can be utilized by other nodes. In addition these benefits, upgrading memory requires only replacing the memory blade [21], i.e., the place where all memories are placed. While disaggregated memory systems are a promising direction for designing future computing systems, a lot of factors need to be examined and researched to properly evaluate and understand disaggregated memory systems.

Processors require extremely fast access to memory. Accessing remote memory through a network increases the delay in accessing the memory and will impact application performance. Furthermore, contention on the centralized memory occurs when multiple nodes are issuing requests to the system memory. Instead of full memory disaggregation, it is better to retain a portion of the memory on the node where it can be managed locally and treated as fast temporary memory [12]. As mentioned earlier, modern applications deal with large amount of data that can exceed the size of the local memory. Accordingly, new memory allocation and page migration policies are warranted on such systems. Furthermore, the cost of unmapping and mapping pages while managing memory should be considered carefully when designing disaggregated memory systems.

To facilitate research efforts in disaggregated memory systems, we propose a disaggregated memory emulation environment that can take into consideration many important system-level aspects. The Structural Simulation Toolkit (SST) [24] has been proven to be one of the most reliable simulators for large-scale systems due to the scalability and modular design of its components. This makes SST the perfect candidate for simulating disaggregated memory systems at scale. One of the current limitations of SST is the lack of a centralized memory management entity that correctly models page faults and requests for physical frames from the simulated machine. Such a limitation becomes more relevant when there are a large number of shared resources (pools). A centralized memory management entity for disaggregated memory, *Opal*

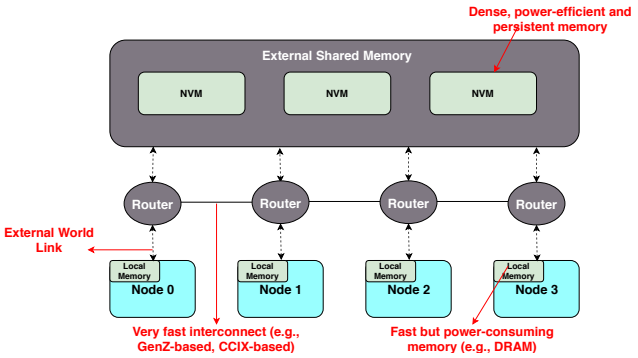


Figure 2: An example of a disaggregated memory system. The system has several nodes (SoCs) where each node may have its own internal memory but share external memory.

[20], was developed that can be used to investigate memory allocation policies, page placement, page migration, the impact of TLB shootdown, and other important aspects that are related to managing disaggregated memory systems. In this paper, we will describe Opal and the different use cases and studies that can leverage our framework.

Previous research on disaggregated memory was conducted on real systems and trace-driven simulators [25, 28]. Trace-driven simulations oversimplify the impact of system-level operations and the out-of-order nature of processing cores and memory systems. They are rarely scalable beyond couple of cores even with very simplified memory and processor models. Moreover, it is difficult to model disaggregated memory as it requires multiple nodes to be simulated at the same time. Real-system prototyping takes a significant amount of time and limits the conclusions to the available hardware and software stack, which reduces the flexibility of design exploration. In contrast, our model is a discrete-event simulation model that is modular and easy to customize.

The rest of the paper is organized as follows. Section 2 presents the background. Section 3 and Section 4 discusses the design and evaluation of our implementation. We conclude the paper in Section 5.

2 BACKGROUND

Many major vendors are considering system designs that utilize disaggregated memory, which can be accessed by a large number of processing nodes. Figure 2 depicts an example of a disaggregated memory system. As shown in Figure 2, the nodes must access an off-chip network to access the external memory. Although local updates to external memory locations can be made visible to all other nodes, scaling the coherence protocols are challenging. While using directories could help, there are still inherent design and performance complexities that can arise. One direction that the vendors will likely adopt is to rely on software to flush updates on local caches to the shared memory and make it visible to other nodes; one can think of it as having a mutex around the shared data, and not releasing it until all the updates

have been flushed to the external memory. Once the lock is released, the other nodes need to make sure they are reading the data from the external memory rather than their internal caches. One way to do that is to use `clflush`[1] after any reads or updates, which guarantees any copies of that memory are invalidated in the cache hierarchy. Another case is where the memory is partitioned between nodes. In this case, each node broadcasts all of its updates and flushes, after which an aggregator node can read the updated values from the external memory. In much simpler cases, such as a file containing a large social network graph where no updates are expected to that graph (read-only), there is no need for special handling of accesses to the graph.

Because off-node memory accesses are expensive, page migration will become a frequent operation on heterogenous and disaggregated memory systems [5, 22]. During page swapping, physical addresses assigned to the virtual addresses can change, hence page table entry (PTE) update is required. The core initiating a PTE update needs to send Inter-Processor Interrupt (IPI) to other cores to force them to invalidate any copies from the updated PTE on their Translation Lookaside Buffers (TLBs). This process is called *TLB shootdown* [26]. To reduce the costs of such an interruption, several TLB shootdown optimization algorithms have been proposed [3, 5] Before these systems can be deployed, it will be important to analyze the impact the page migration will have on disaggregated memories.

Bandwidth and speed of the memory are the main parameters that decide the reliability and performance of memory-intensive applications. Disaggregated memory systems can provide better bandwidth by scaling the number of channels to shared memory but this often comes at the expense of latency. For instance, blade servers [21] were proposed with memory blades that use fast shared communication networks and contain racks of memory modules.

To the best of our knowledge there is currently no simulation platform that can properly simulate and model disaggregated memory systems. Several disaggregated memory performance emulators have been developed [4, 11, 16], which divide physical memory to evaluate the remote memory. For these tests, remote memory latency is emulated through a device driver. Unfortunately, relying on real-system emulation restricts the conclusions and design space exploration to a narrow space that is constrained by the real-system configurations.

3 DESIGN

The Structural Simulation Toolkit (SST) is an architectural simulator that is known for its scalability and reliability due to its modular design and parallel nature. Implementing a disaggregated memory system design in SST opens up opportunities to explore and examine many challenges. Disaggregated memory systems require global memory managers to handle the system shared memory, initiate and broadcast TLB shootdown requests, implement page migration and allow for sharing memory between nodes. To model these aspects of the system, we propose Opal, a centralized memory manager that is implemented as a part of SST to help researchers in studying the functionalities, bottlenecks and optimizations for managing disaggregated memory systems. For the rest of this section, we describe the Opal component and how it can be utilized to investigate disaggregated memory systems.

3.1 Opal

Opal can be thought of as the Operating System (OS) memory manager and, in the case of a disaggregated memory system, the system memory allocator/manager. In conventional systems with a single level memory, once a process tries to access a virtual address, a translation is triggered to map the virtual address to a physical address. If a translation is not found, and the hardware realizes that either there is no mapping to that virtual address or the access permissions would be violated, it triggers a page fault that is handled by the OS. The page fault handler maps the virtual page to a physical page that is chosen from a list of free frames (physical pages). Once a physical page is selected, its address is inserted in the page table along with the corresponding access permissions. Any successive accesses to that virtual address will result in a translation process that concludes with obtaining the physical address of the selected page. Since SST aims for fast simulation of HPC systems, it does not model the OS aspects of this sequence of events. However, the memory allocation process will have a major impact on performance for heterogeneous memory systems and disaggregated memory, simply because of the many allocation policies that an OS can select from. Moreover, allocation policies are not well understood on disaggregated memory systems, making it important to investigate them to discover the best algorithm or heuristics to be employed for both performance and energy efficiency. Opal is proposed to fill this role; facilitating fast investigation and exploration of allocation policies in heterogeneous and disaggregated memory systems.

Each component in SST typically represents a subsystem in a real system. SST models a wide range of components such as cores, MMU units, memory hierarchy, routers, and different memory models like DRAM and NVM. Components are ticked according to the component clock frequency set up during configuration. Links are used to communicate between components. Each link can be configured with a latency. We used the Ariel, Samba [6], Messier [7] and Merlin components in SST to simulate CPU cores, MMU unit, NVM memory and network respectively to implement disaggregated memory system design with the help of Opal component.

As shown in Figure 3, Opal and external memory are maintained remotely and each node is connected to Opal and external memory through external links. Processing cores and memory management units are connected to global memory manager, Opal. In our design, we maintained an internal

[1] Instruction which helps in flushing the cache contents of a process from the userspace.

router that helps in communicating between cache and memory components. Likewise, an external router is maintained to connect the internal router with external memory through a network bridge that has its latency modelled after GenZ [14]. This way, communication between nodes and external memory takes place though internal and external routers. To make it realistic, links to external memory is configured with high latency and links to internal memory is configured with low latency.

Processing cores are connected to Opal to pass hints about memory allocations. For instance, calls to `malloc` or `mmap` do not immediately allocate physical pages, but are allocated at the time of mapping, during page fault. Opal can use hints sent from cores to decide where to allocate the physical page. This is similar to libNUMA malloc hints, which will be recorded and used later by the kernel at the time of on-demand paging. CPU cores can trigger TLB shootdown events to all the other cores, including cores on other nodes. It is cumbersome to create links between each core to send events like TLB shootdown. Hence, we facilitate a communication medium between nodes through Opal. CPU cores communicate with Opal, sending TLB shootdowns events, using a core to Opal link.

The hardware MMU units have links to Opal, so that once a TLB miss and page table walk conclude with a page fault request (unmapped virtual address), a request for physical frame allocation is sent to Opal. Allocation requests come from the page table walker when the accessed virtual page has never been mapped, which resembles the minor page fault and on-demand paging on the first access to virtual pages in real systems. Opal searches for any hints associated with the page fault. If the hints are available, memory is allocated according to the hints from a specific memory region, if not, Opal checks for free frames according to the allocation policies, described in Section 3.3, and allocates a frame to the corresponding memory request. Apart from this, during TLB shootdown, Opal sends invalid addresses to all the MMU's through the MMU unit to Opal link and the MMU unit responds with an acknowledge event to Opal after invalidating the addresses.

Hence, Opal must be connected to both a MMU unit, such as Samba, for receiving page fault requests and a processing element, such as Ariel. To allow this, Ariel cores and Samba units should connect to their respective ports in Opal, *coreLink_n* and *mmuLink_n*. For example, *coreLink_0* port of Opal can be connected to *opal_link_0* port of Ariel core and *mmuLink_0* port of Opal can be connected to *ptw_to_opal0* port of Samba.

Before diving into the details of Opal, we will start with discussing different ways of managing disaggregated memory systems:

3.1.1 Exposing External Memory Directly to Local Nodes. In this approach a local node OS (or Virtual Machine) sees both the local memory and external memory, however, it needs to request physical frames from a central memory manager to be able to access external memory legitimately. To enforce access permission, and to achieve isolation between

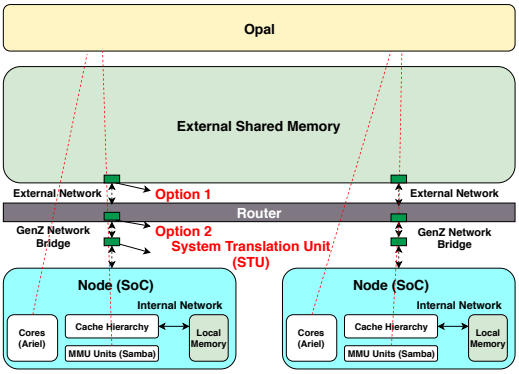


Figure 3: A simulated system that uses Opal for centralized memory management.

data belonging to different nodes/users, the system must provide a mechanism to validate the mappings and the validity of physical addresses being accessed by each node. To better understand the challenges of this scheme, Figure 3 depicts different options to implement access control on shared resources in such management scheme.

As shown in the Figure 3, Option ① would be to check if the requesting node is eligible to access the requested address at the memory module level. This implementation requires a bookkeeping mechanism at the memory module level (or in the memory blade) to check the permission of every access. If the access is valid, then the request will be forwarded to the memory, otherwise either a random data is returned or an error packet (access violation) is sent back to the requesting core. Since the external memory is shared between nodes, the system memory manager must have a consistent view of allocated pages and their owning nodes. One way to implement this is through a device driver (part of the local nodes' OS) that can be used to communicate, either through the network or predefined memory regions, with the external memory manager. Option ② is similar but instead of relegating the permission check to the memory module, the router will have mechanisms to check if the accessed physical addresses are granted to the requesting node. In both the options, nodes will not be able to have a direct access or modifications for such permission tables, only the system memory manager will have such access. Such guarantee can be implemented by encrypting requests with some integrity and freshness verification mechanisms. There are many benefits of these schemes, such as: page table walking process is not modified and it is much faster than virtualized environments (4 steps vs. 26 steps). Also, node-level memory manager optimizations and page migrations are feasible (unlike virtualized environments). But the operating system must be patched with a device driver to communicate with external memory manager and the centralized memory manager becomes a bottleneck if not scalable.

3.1.2 Virtualizing External Memory. In this approach, each node has the illusion that it owns all of the system memory. In fact, in this scheme, the OS doesn't need to be aware

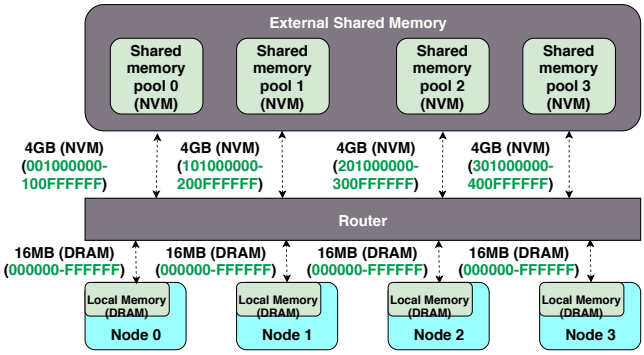


Figure 4: Example configuration

of the current state of the actual system physical memory. Figure 3 depicts the virtualized system memory scheme.

As shown in the Figure 3, the system translation unit (STU) must be added to support translation from the *node physical address* to the *system physical address*. The STU can be implemented as an ASIC-based or FPGA-based unit that takes a physical address from the node and translate it into the corresponding system physical address. In case the address has never been accessed, an on-demand request mechanism is initiated by the STU to request system physical page. The STU might need to do a full system page table walk to obtain the node to system translation. Most importantly, the STU can be updated only through the system memory manager. This scheme is better if OS does not need to be changed. But the STU will need to walk the system level page table in addition to walking the node's page table at the node level. Also, there is no guarantee of where the system physical pages that back up the node physical pages exist.

3.2 Opal Configuration

Opal should be configured with component-specific, node-specific and shared memory-specific information as shown in Table 1. Component -specific information includes clock frequency, maximum instructions per cycle, etc. Node specific information includes number of nodes, number of cores per node, clock frequency per node, per node network latency to access Opal component, node memory allocation policy as explained in section 3.3 and local memory information. Shared memory-specific information includes number of memory pools that shared memory is divided into and the respective memory pool parameters. Both per-node local memory and per-shared memory pool parameters are related to memory and they are explained separately in Table 2. Each of these parameters should be appended with memory related parameters as shown in Table 1. Table 2 describes the memory pool-specific parameters. Each memory pool, whether shared or local, needs a starting address, pool size, frame or page size, and memory technology.

We show a basic configuration used to test a disaggregated memory system with Opal in Figure 4. The respective parameters can be found online [1]. According to the example configuration, the clock frequency of Opal is $2GHz$ ("*clock*" : "$2GHz$"). In every cycle Opal can serve up to 32 requests ("*max_inst*" : 32). The system has 4 nodes ("*num_nodes*" : 4) with a private memory each and shared global memory is divided into 4 memory pools ("*shared_mempools*" : 4). Each node has 8 cores ("*node*0.*cores*" : 8) with a clock frequency of $2GHz$ ("*node*0.*clock*" : "$2GHz$") per core. The private memory uses $DRAM$ ("*node*0.*memory.mem_tech*" : 0) technology with a size of $16MB$ ("*node*0.*memory.size*" : 16384), a starting address of 0 ("*node*0.*memory.start*" : 0), and a frame size or page size of 4KB ("*node*0.*memory.frame_size*" : 4). The total global or shared memory is $16GB$, which is divided into 4 memory pools each of $4GB$ ("*shared_mem.mempool*0.*size*" : 4194304). The starting address of shared memory pool 0 is 001000000 ("*shared_mem.mempool*0.*start*" : 001000000) which is equivalent to local memory(16MB) + 1; the starting address of memory pool 1 is 101000000 ("*shared_mem.mempool*1.*start*" : 101000000), which is equal to the starting address of shared memory pool 0 + shared memory pool 0 size. Figure 4 shows the starting address of each memory pool, from which the size of each memory pool can

Table 1: Opal Parameters

Parameter	Description
clock	frequency of Opal component
max_inst	maximum instructions processed in a cycle.
num_nodes	number of nodes.
node_i_cores	number of cores per node.
node_i_clock	frequency of each node.
node_i_latency	latency to access Opal component per node.
node_i_allocation_policy	memory allocation policy per node.
node_i_memory.	local memory-specific information per node. These come under memory parameters and are shown in Table 2
shared_mempools	number of shared memory pools to maintain shared memory.
shared_mem.mempool_i.	global memory-specific information per shared memory pool. These come under memory parameters and are shown in Table 2

Table 2: Memory Pool Parameters

Parameter	Description
start	starting address of the memory pool.
size	size of the memory pool in KB's.
frame_size	frame size of each frame in memory pool in KB's. This is equivalent to page size.
mem_tech	memory pool technology ($0 : DRAM, 1 : NVM$).

be deduced. Each shared memory pool is of NVM type ("$shared\_mem.mempool0.mem\_type$" : 1) with a frame size of $4KB$ ("$shared\_mem.mempool0.frame\_size$" : 4). Memory allocation policy used for all the nodes is alternate memory allocation policy ("$node0.allocation\_policy$" : 1) which is explained in the Section 3.3. The network latency to communicate with Opal is 2 micro seconds ("$node3.latency$" : 2000).

3.3 Memory Allocation Policies

Multiple memory allocation policies are implemented in our design, which are described below.

3.3.1 Local Memory First Policy: Local memory is given more priority than shared memory, that is, memory is searched in local memory and if local memory is full then shared memory is searched for memory. If shared memory is spread into different memory pools, then a shared memory pool is chosen randomly among different memory pools until some space is found. If none of the memory pools are available, that is total memory is full, then an error message is thrown. This memory allocation policy can be chosen by setting "$allocation\_policy$" parameter of a node to 0.

3.3.2 Alternate Memory Allocation Policy: For every two memory requests, one frame is allocated from local memory and the other from shared memory. For example, if two shared memory pools are maintained, then the 1st time memory is allocated from local memory, memory for the 2nd request is assigned from shared memory pool 1, memory for the 3rd request is assigned from local memory, memory for the 4th request is assigned from shared memory pool 2 and so on. This memory allocation policy can be chosen by setting "$allocation\_policy$" parameter of a node to 1.

3.3.3 Round Robin Memory Allocation Policy: Memory frames are scheduled to be allocated from shared and local memory based on the total number of memory pools, which includes local memory pool of a node and total shared memory pools in a round robin fashion. If two shared memory pools are maintained, then for the 1st memory request, memory is allocated from local memory, for the 2nd memory request, memory is allocated from shared memory pool 1, for the 3rd memory request, memory is allocated from shared memory pool 2, for the 4th memory request, memory is allocated from local memory and so on. This memory allocation policy can be chosen by setting "$allocation\_policy$" parameter of a node to 2.

3.3.4 Proportional Memory Allocation Policy: The proportion at which memory frames are allocated from shared and local memory is based on the fraction of local memory size to total shared memory size. For example, if the local memory size is 2GB and shared memory size is of 16GB, then, for the 1st memory allocation request, memory is allocated from local memory while the next 8 memory requests are allocated from shared memory in sequential order. For the next memory request, which is 10th memory request, memory is allocated from local memory and so forth. This memory allocation policy can be chosen by setting "$allocation\_policy$" parameter of a node to 3.

3.4 Communication Between Nodes

Opal also allows nodes to communicate directly with one another by sending hints with same $fileID$ to Opal using Ariel $ariel\_m$ map_mlm and $ariel\_mlm\_malloc$ calls. Opal checks if the received $fileID$ is registered with any memory. If it is, then the specific page index is sent to the requesting node. If the $fileID$ is not registered with any memory page, then memory is allocated based on the requested size. The allocated memory region is now registered with the requester $fileID$. Nodes can share information just by writing information to the specific pages. This reduces costly OpenMPI calls to share information between nodes.

4 EVALUATION

We validated our design by calculating the performance of the system in-terms of instructions per cycle (IPC). We vary the number of nodes, number of shared memory pools and memory allocation policies. The average number of instructions per cycle is taken into consideration. Simulation parameters and applications that we used along with application parameters are shown in Tables 3 and 4 respectively.

Table 3: Simulation Parameters

Element	Parameters
CPU	8 Out-of-Order cores, 2GHz, 2 issues/cycles, 32 max. outstanding requests
L1	private, 64B blocks, 32KB, LRU
L2	private, 64B blocks, 256KB, LRU
L3	shared, 64B blocks ,16MB, LRU
Local memory	2GB, DDR4-based DRAM
Global memory	16GB, NVM-based DIMM (PCM), 128 max. outstanding requests, 16 banks $300ns$ Read Latency, $1000ns$ Write Latency
External network latency	20ns[10]

Table 4: Applciations

Application	Value
XSBench [27]	-s large -t 8
Lulesh [19]	-s 120
SimpleMoC [17]	-t 8 -s
Pennant [15]	leblancbig.pnt
miniFE [18]	-nx 140 -ny 140 -nz 140
NAS:IS [8, 9]	class C

Table 3 depicts simulation parameters for our experiments. According to this, each node has 8 cores and each core can serve up to 2 instructions per cycle. The clock frequency of the cores is 2GHz. Each core is configured to service up to 100 million instructions. Three levels of cache are used, L1, L2,

and L3, with sizes of 32KB, 256KB and 16MB respectively and each are of non-inclusive type. Local memory size is 2GB and is of DRAM type. External memory is of NVM type with 16GB size. Network latency is critical in disaggregated memory system. For the external network latency, we use $20ns$ which has been modelled after the GenZ network latency.

Since our focus is on HPC applications we evaluated our design using 6 HPC mini applications. XSBench [27], a mini-app representing a key computational kernel of the Monte Carlo neutronics application, OpenMC. Lulesh [19], a mini-app for hydrodynamics. Pennant [15] is an unstructured mesh physics mini-app designed for advanced architecture research. SimpleMOC [17], mini-app is to demonstrate the performance characteristics and viability of the Method of Characteristics (MOC) for 3D neutron transport calculations in the context of full scale light water reactor simulation. NASA IS [8, 9] mimic the computation and data movement characteristics of large scale computational fluid dynamics (CFD) applications. IS is an integer sort kernel which performs a sorting operation. MiniFE [18] is a proxy application for unstructured implicit finite element codes. Applications and their parameters are shown in Table 4. We decided upon these specific applications as these are memory intensive.

4.1 Memory Allocation Policies

If more memory is allocated from shared memory, the performance of the system worsens as the delay in accessing shared memory is high. Memory allocation policies, explained in section 3.3, control allocation of local and shared memory. Contention at shared memory is one of the key factors that contributes to the performance in disaggregated memory systems. The more the contention at the memory, the more will be the delay in accessing memory. Contention at memory is higher if more nodes are accessing memory at a given time. Accordingly, we observed the following traits for each memory allocation policy.

4.1.1 Local Memory First (LMF) Policy: According to local memory first allocation policy, memory is allocated in private memory first and if there is no space in private memory then memory is allocated from global memory. The benchmark applications that we used occupy a maximum of approximately 500MB of memory to generate 100 million instructions. Because each node has its private memory of 2GB, all of the memory pages should be allocated form local memory and the performance of the nodes should be same as there is no contention at local memory due to other nodes. Our results in Figure 5(a) reflect this. Irrespective of the number of nodes and number of shared memory pools the performance of each node, i.e., number of instructions per cycle is equal. We show this to understand the memory intensity of the benchmarks. According to Figure 5(a), the IPC of XSBench and MiniFE is around 0.6. Lulesh, Pennant and NAS:IS have an IPC of around 1.1. Wherein the IPC of SimpleMoC is 1.6. From this it can be understood that XSBench and MiniFE are more memory intensive, SimpleMoC is less memory intensive, and Lulesh and Pennant are moderately memory intensive among the set of benchmarks that we experimented with.

4.1.2 Alternate (ALT) Policy: In this memory allocation policy, for every other page fault, a page is allocated from the shared memory. Accordingly, almost half of the pages are from the shared memory, i.e., among 500MB of memory that the applications use, 230MB of memory is from shared memory. From Figure 5(a) the IPC of Lulesh is 1.1 while Figure 5(b) shows an IPC of 0.2. The performance decreases by 81% when shared memory is used. It further decreases if there are a greater number of nodes accessing the shared memory. For the same benchmark, the IPC is 0.05 when 4 nodes share the external memory. As the number of nodes making use of the shared memory increases, contention at the shared memory increases and the individual node performance decreases. Contention can be reduced by dividing the shared memory into number of memory pools and hence the performance of the system increases. From Figure 5(b) it can be seen that for Pennant, the IPC is 0.12 and 0.34 with 1 node when shared memory is maintained in 1 shared memory pool and 4 shared memory pools respectively. With 4 nodes, the IPC is 0.1 when shared memory is maintained in 4 shared memory pools, which is almost equivalent to the performance of the system with 1 node when shared memory is maintained in only 1 shared memory pool.

4.1.3 Round Robin (RR) Policy: Memory is allocated based on the number of shared and local memory pools. The more the number of shared memory pools, the more memory addresses are allocated from the shared memory and the performance decreases. Figure 5 shows that, for the 4 node SimpleMoC benchmark with 4 shared memory pools, the IPC is 0.15 for the RR policy and 0.25 for the ALT policy. This is due to more memory is allocated from shared memory in the RR memory allocation policy with 4 shared memory pools. When shared memory is maintained only in 1 memory pool, RR memory allocation policy is same as ALT memory allocation policy. From Figure 5(c) it can also be observed that, for some applications, when shared memory is maintained in more shared memory pools, the performance decreases due to more memory being allocated from shared memory. For instance, the IPC of XSBench drops from 0.28 to 0.23 when shared memory is divided into 4 shared memory pools compared to when shared memory is maintained in 1 shared memory pool.

4.1.4 Proportional (PROP) Policy: Memory allocation in based on the proportion of local and shared memory. From the configuration that we used, 16GB of shared memory and 2GB local memory, the proportion at which shared and local memory are allocated is 8:1, i.e., for every 9 memory allocations 8 memory allocations are from shared memory and 1 memory allocation is from local memory. According to this, more memory is allocated from shared memory in comparison with RR and ALT allocation policies. From Figure 5(d) it can be clearly observed that, for miniFE, the IPC is 0.06 with 1 node and when shared memory is maintained in only

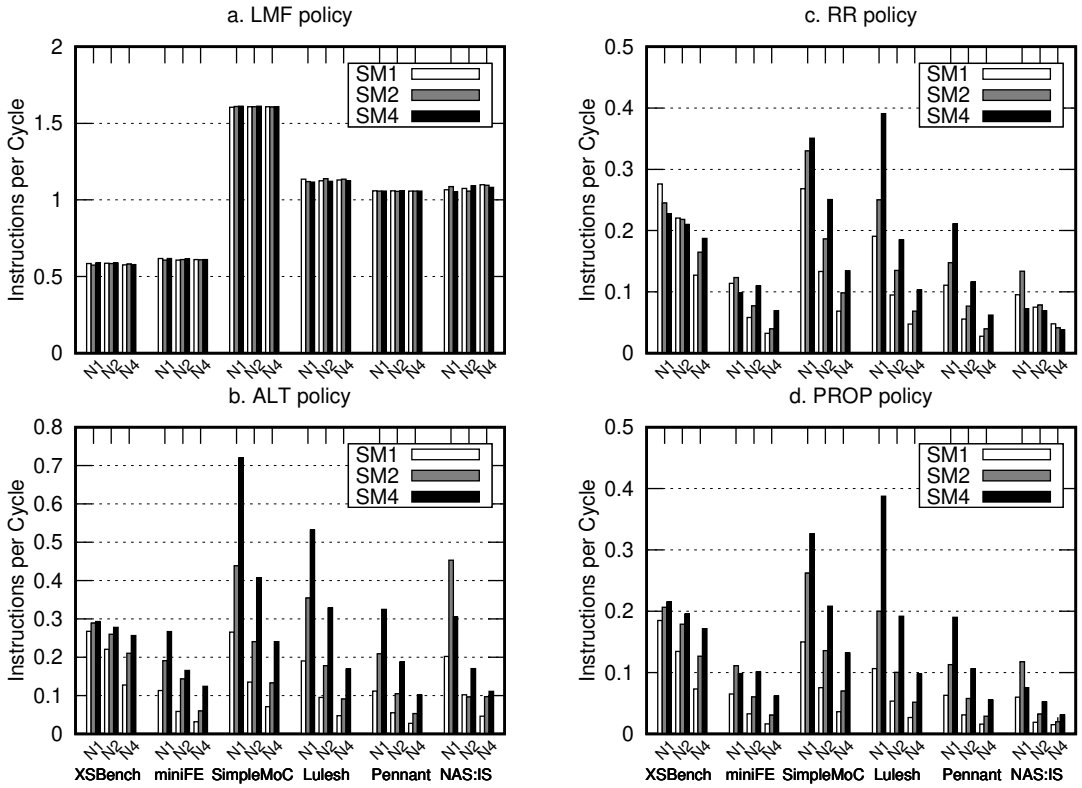


Figure 5: Performance in instructions per cycle of disaggregated memory system with different memory allocation policies. N indicates number of nodes. SM indicates number of shared memory pools. LMF, ALT, RR and $PROP$ indicate local memory first, alternate memory, round robin and proportional memory allocation policies.

1 shared memory pool. This is less than the IPC of ALT memory allocation policy and RR memory allocation policy which is around 0.11 each, from Figures 5(b) and 5(c). As the nodes increased from 1 to 4, the IPC of the system further decreased from 0.06 to 0.02. When shared memory is divided into 4 shared memory pools the IPC of the system increased to 0.06.

We observe that dividing shared memory into more shared memory pools does not always improve the performance of the system. The performance depends on the application characteristics as well[2]. From Figure 5 it can be seen that for NAS:IS benchmark, for several memory allocation policies, the IPC, when shared memory is maintained in 2 shared memory pools, is more when compared with IPC of the system when shared memory is maintained in 4 shared memory pools with 1 node in the system. We suspect that NAS:IS is latency sensitive and performs better when local memory is used even though it has limited memory-level parallelism. When shared memory is divided into 2 shared memory pools, this can lead to increase in number of memory accesses serviced by global memory as in round-robin allocation policy, however, this can also improve the bandwidth and memory-level parallelism. Meanwhile, increasing the number of pools to 4 can lead to performance degradation as the increase in memory access latency due to accessing global memory is no longer amortized by the increase in bandwidth.

[2]Accordingly our future work involves developing application aware memory allocation policies

5 CONCLUSION

Disaggregated memory is a promising memory architecture to take advantage of modern memory technologies, for sharing data, and to efficiently utilize memory. While it may be a useful system design, before fully adopting this architecture, there are a lot of challenging design parameters that must be fully understood such as speed, memory management policies, virtual to physical address translation, page migration, and quality of service. To this end, we proposed a new disaggregated memory emulator model to examine and explore various aspects related to disaggregated memory architecture. Specifically, we implemented a centralized memory manager in SST which has capability to manage memory in disaggregated memory systems as well as in general systems. Our future work involves implementing and exploring various page migration methodologies and enforcing QoS in disaggregated NVM memory systems.

ACKNOWLEDGMENTS

This work has been funded through Sandia National Laboratories (Contract Number 1844457) Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] [n. d.]. https://github.com/VamseeReddyK/Opal_example/blob/master/example_configuration.txt

[2] 2015. Adaptive Resource Optimizer For Optimal High Performance Compute Resource Utilization. Synopsys Inc, silicon to software, Mountain View, 1–5.

[3] Nadav Amit. 2017. Optimizing the TLB shootdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. 27–39.

[4] Krste Asanovic and David Patterson. 2014. Firebox: A hardware building block for 2020 warehouse-scale computers. In *USENIX FAST*, Vol. 13.

[5] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2017*. IEEE, 273–287.

[6] A Awad, SD Hammond, GR Voskuilen, and RJ Hoekstra. 2017. *Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework*. Technical Report. Technical Report SAND2017-0002, Sandia National Laboratories, Albuquerque, NM.

[7] Amro Awad, Gwendolyn Renae Voskuilen, Arun F Rodrigues, Simon David Hammond, Robert J Hoekstra, and Clayton Hughes. 2017. *Messier: A Detailed NVM-Based DIMM Model for the SST Simulation Framework*. Technical Report. Technical Report. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).

[8] David H Bailey. 2011. Nas parallel benchmarks. In *Encyclopedia of Parallel Computing*. Springer, 1254–1259.

[9] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[10] Shekhar Borkar. 2006. Networks for multi-core chips–a controversial view. In *Workshop on on-and off-chip interconnection networks for multicore systems (OCIN), Stanford*.

[11] Dhantu Buragohain, Abhishek Ghogare, Trishal Patel, Mythili Vutukuru, and Purushottam Kulkarni. 2017. DiME: A Performance Emulator for Disaggregated Memory Architectures. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM, 15.

[12] Daniel Turull Chakri Padala and Vinay Yadav. 2017. Time for memory disaggregation? Ericsson Research Blog. *Online]. https://www.ericsson.com/research-blog/time-memory-disaggregation/* (may 2017).

[13] Dan Comperchio and Jason Stevens. 2014. Emerging Computing Technologies: Hewlett-Packard's "The Machine" Project. In *HP Discover 2014 conference held in Las Vegas June 10-12*. Willdan Energy Solutions, 1–4.

[14] Gen-Z Consortium et al. 2017. Gen-Z–A New Approach to Data Access.

[15] Charles R Ferenbaugh. 2015. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 4555–4572.

[16] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap.. In *NSDI*. 649–667.

[17] Geoffrey Gunow, John Tramm, Benoit Forget, Kord Smith, and Tim He. 2015. Simplemoc-a performance abstraction for 3d moc. (2015).

[18] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574* 3 (2009).

[19] Ian Karlin, Jeff Keasler, and JR Neely. 2013. *Lulesh 2.0 updates and changes*. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[20] Vamsee Kommareddy, A Awad, C Hughes, and SD Hammond. 2018. *Opal: A Centralized Memory Manager for Investigating Disaggregated Memory Systems*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[21] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 267–278.

[22] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *21st International Symposium on High Performance Computer Architecture (HPCA), 2015*. IEEE, 126–136.

[23] Hugo Meyer, Jose Carlos Sancho, Josue V Quiroga, Ferad Zyulkyarov, Damian Roca, and Mario Nemirovsky. 2017. Disaggregated computing. an evaluation of current trends for datacentres. *Procedia Computer Science* 108 (2017), 685–694.

[24] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, E CooperBalls, et al. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (2011), 37–42.

[25] Stcphcn W Sherman and JC Browne. 1973. Trace driven modeling: Review and overview. In *Proceedings of the 1st symposium on Simulation of computer systems*. IEEE Press, 200–207.

[26] Patricia J. Teller. 1990. Translation-lookaside buffer consistency. *Computer* 23, 6 (1990), 26–36.

[27] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[28] Richard A Uhlig and Trevor N Mudge. 1997. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)* 29, 2 (1997), 128–170.

Heterogeneous Memory and Arena-Based Heap Allocation

Sean Williams
New Mexico Consortium
swilliams@newmexicoconsortium.org

Latchesar Ionkov
Los Alamos National Laboratory
lionkov@lanl.gov

Michael Lang
Los Alamos National Laboratory
mlang@lanl.gov

Jason Lee
Los Alamos National Laboratory
jasonlee@lanl.gov

ABSTRACT

Nonuniform Memory Access (NUMA) will likely continue to be the chief abstraction used to expose heterogeneous memory. One major problem with using NUMA in this way is, the assignment of memory to devices, mediated by the hardware and Linux OS, is only resolved to page granularity. That is, pages, not allocations, are explicitly assigned to memory devices. This is particularly troublesome if one wants to migrate data between devices: since only pages can be migrated, other data allocated on the same pages will be migrated as well, and it isn't easy to tell what data will be swept along to the target device. We propose a solution to this problem based on repurposing arena-based heap management to keep locality among related data structures that are used together, and discuss our work on such a heap manager.

ACM Reference format:
Sean Williams, Latchesar Ionkov, Michael Lang, and Jason Lee. 2018. Heterogeneous Memory and Arena-Based Heap Allocation. In *Proceedings of MCHPC'18: Workshop on Memory Centric High Performance Computing, Dallas, TX, USA, November 11, 2018 (MCHPC'18),* 5 pages.
DOI: 10.1145/3286475.3286568

1 INTRODUCTION

There have been some steps recently to incorporate heterogeneous memory into high-performance computing. Most prominently, the now-defunct Intel Knights Landing [4] included integrated higher-bandwidth memory. Likewise, Nvidia is working to unify CPU and GPU memory with its NVLink[8] fabric and include higher-bandwidth memory, and Intel and others are bringing non-volatile memory to DIMM slots in order to have a higher-capacity, lower-performance option that is integrated into the memory address space.

The Intel Knights Landing exposes its high-bandwidth memory to the user as a NUMA node as do the IBM CORAL [7] systems. This makes perfect sense, since NUMA is a preexisting facility for bridging the gap between physical and virtual memory—in principle, virtual memory removes the need to care about physical devices. Thus, we expect that NUMA will continue to be used as the first-order abstraction for heterogeneous memory. In the past NUMA distance has just been representative of the "hop" distance between a CPU and a memory node. With high-bandwidth memory the NUMA distance is being used as a way to differentiate types of heterogeneous memory. Nevertheless, both multisocket machines (the traditional NUMA use case) and heterogeneous memory reassert the importance of knowing and specifying a physical home for a page of memory.

In Linux, users can interact with NUMA-informed placement via memory policies. The chief policies are, attach this page to the closest NUMA node; try to attach this page to a specific node, and if it's full, attach to the closest node; and attach this page to a specific node, and fail if it's full. This situation opens up many both practical and theoretical problems of varying importance and tractability—there's a large conceptual gap between these simple policies and the actual use cases of heterogeneous memory. This paper is about one of the simpler, more practical ones: that all these policies operate at page granularity.

The canonical bridge between pages and data structures is the heap manager, which dices up pages in such a way as to balance efficient utilization of memory with the performance of the allocator. But part of the point of heterogeneous memory is the fundamental trade-off between speed and capacity, so one would expect optimal use of such memory systems to involve a lot of churn. On the other hand, since memory can only be moved between NUMA nodes in page-sized chunks, an obvious problem arises: what's good for the data structure may not be good for the page. If a page contains both "hot" and "cool" data (i.e., data that would benefit from residing in high-performance memory, along with data that wouldn't), then it's never clear what the right choice is: should the page be on a high-performance device, wasting some of this precious resource, or should it be on a normal-performance device, to make more efficient use of limited space?

There's a potential answer in the concept of *arenas*, in which a heap manager maintains multiple heaps. This was traditionally done to reduce lock contention: heaps require substantial bookkeeping, and the associated data structures can need updating when servicing `malloc` and `free` calls. Having one or more heaps per thread can reduce or eliminate contention for each heap's bookkeeping data.

A major drawback to the current situation, where pages are tied to NUMA nodes, is that it can be hard to assess the value of transferring a page to or from a high-performance memory device. Our proposal is that, rather than using arenas for the purpose of reducing lock contention, arenas be used to group data structures that should "travel together." Under this proposal, the assessment is not, "is it a good idea to transfer this data structure"? but, "is it a good idea to transfer this whole arena"? The remainder of this paper

MCHPC'18, Dallas, TX, USA

978-1-4503-6113-2/18/11…$15.00
DOI: 10.1145/3286475.3286568

will be devoted to discussing various aspects, and pros and cons, of this approach. Additionally, we provide some use cases for our implementation and results from experiments.

2 APPROACH

2.1 Homogeneity

NUMA was designed for multisocket machines, so it is based on an assumption that the only difference between memory devices is latency. Thus, the default NUMA policy, in which allocations are preferentially located on the memory node nearest the allocating processor, makes perfect sense. This point is even baked into the name, nonuniform memory *access*: the assumption is that memory devices are homogeneous, and only their access is different.

With heterogeneous memory, it is typically the case that both the characteristics (e.g., performance) and the access of memory devices will be different. This makes it difficult to determine what the right allocation policy is and, whether there actually is a right allocation policy. Heterogeneity makes the problem of allocation placement far more difficult than the merely nonuniform-access situation.

In essence, we can reduce this problem to three components: mapping allocations to pages, mapping pages to policies, and mapping policies to devices.

2.2 Allocations → Pages: Arenas

At the operating system level, memory is only available to page granularity, via the `mmap` system call. As a consequence, memory can only be attached to devices (i.e., NUMA nodes) at page granularity. Of course, most data structure sizes are not multiples of a page size (4 KiB, 2 MiB, etc.), so we use heap managers (i.e., `malloc`) to pack data onto pages in a reasonably efficient way. But this is another situation where a traditional assumption of homogeneity becomes pernicious.

But what relationship should be established between pages, allocations, and devices? Intel's `memkind` [3] library allows one to create heaps associated with particular "kinds" of memory, e.g., high-bandwidth. The basic idea of `memkind`, that it's necessary to address the problem of matching allocations to heterogeneous pages using an arena-based heap allocator, is sound.

What role should arenas serve in solving this problem? In `memkind`, the intention seems to be that arenas enumerate the kinds of memory, so that one has a default arena, a high-bandwidth arena, a high-capacity arena, and so on.

As will be discussed in Section 2.3, we believe that the ability to move allocations between devices is an important capability. How does one move allocations? The `memcpy` function puts the allocation at a new address, so preexisting pointers to the allocation will continue to point to its old address. On the other hand, the `migrate_pages` system call preserves pointers, but as the name suggests, it only operates on whole pages.

This has the obvious consequence that, if one migrates an allocation, `migrate_pages` moves all pages containing that allocation. Since the exact behavior of `malloc` is hard to predict, this could cause arbitrarily bad data movement to occur: Imagine a data structure was involved in an intensive computation and was on high-performance memory, but that computation is over. In principle, these data should be moved back to normal memory. But it is unclear what else occupies that page and if there is other data on it that are still needed in high-performance memory.

The key dynamic here is, since data must be migrated in page-sized chunks, that arenas should be deployed based on assumptions about the coherence of data structures. If the programmer believes data structures A and B are typically operated on together, then they should be placed on the same page. If data structure C has nothing to do with A or B, then C should not be placed on a page with them.

What this means in practice is, the proposed API allows one to define any number of arenas, and select which arena any particular data structure is placed on. When one wants to migrate data to a new device, one migrates its entire arena. This has the effect of ensuring that migration occurs only with data structures that should migrate together, and excluding data structures that should not be swept along for the ride.

2.3 Pages → Policies: Modeling

The big problem of the previous section is, how does one decide when to move an arena, and to which device? This opens up a vast configuration space, which some "hero programmers" may use for obsessive optimization, but most people won't want to bother. One could imagine treating memory placement as a sort of numerical optimization problem, i.e., one could imagine constructing a model of where each allocation should reside as a function of time.

Consider the form of such a model: $f(\mathbf{A},t,\mathbf{I}) = (\cdots)$, where $\mathbf{A}$ is the set of allocations the program will make, t is time (i.e., the number of instructions previously executed), and $\mathbf{I}$ is an input deck. What is the cardinality of $\mathbf{A}$ for a particular program? In other words, how many allocations does a particular program make? We could simplify this question even further to, how many allocations are made by a call to `malloc` within a loop or recursive function? The answer to that question is well known; unfortunately, the answer is $\perp$, i.e., the value of undecidable computation. Thus, the cardinality of $\mathbf{A}$, the set of allocations made by a program, cannot be known in general, so this modeling exercise would seem to be off to a rough start.

If we instead pose our model in terms of arenas, then this problem goes away: the number of arenas is explicitly chosen by the programmer, so the cardinality of the set of arenas is not dependent on any particular run of the program, much less on all possible runs. Under this arrangement, therefore, we can indeed *pose* a model of memory placement by allowing the programmer to give the problem known bounds.

Notably, this present work only addresses the decidability of $\mathbf{A}$, which makes it possible to pose models. Whether those models will be any good likely hinges on the decidability of f overall, which is a tall order. We can certainly bound this problem by using knowledge of our HPC applications. By categorizing HPC applications we can come up with a set of ad-hoc methods that will improve performance of HPC applications in most cases.

2.4 Policies → Devices: Orderings

Intel's `memkind` allows one to allocate memory on "kinds" of devices, and one kind is high-bandwidth memory. How do its authors decide what constitutes a high-bandwidth NUMA node? Let us begin with a different question: what is NUMA distance?

Remember that NUMA was developed for homogeneous memory that has heterogeneous access. NUMA distance is, not surprisingly, a relative measure of latency. Among other things, this assumes homogeneous capacity, so it always makes sense to choose the lowest-latency memory—there are no trade-offs here.[1]

High-bandwidth memory does represent a trade-off: we can deduce this from the fact that one has even bothered to differentiate them, i.e., to make normal memory in addition to high-bandwidth memory. If high-bandwidth memory were strictly better, then system designers would do away with normal memory. The ordinary trade-off is capacity, so the designers were faced with a conundrum: if high-bandwidth memory is given a low NUMA distance, then allocations will default to it, and bandwidth-insensitive data will fill it up. Special memory requires special allocation, so high-bandwidth memory is given a high NUMA distance.

This means, in order to allocate high-bandwidth memory, one needs to know it's there. Which gets us back around to `memkind`: how does it know that a system has high-bandwidth memory? If you dig into the library, you will eventually find an inline assembly block with `cpuid`, followed by some bitwise operations on magic numbers. These identify whether the CPU family is Intel Knights Landing, and if it is, then high-bandwidth allocations are bound to the NUMA node with the magic distance of 31. Other architectures are not supported—though in Intel's defense, heterogeneous memory is still pretty exotic.

This discussion was intended to motivate the following one: how do we decide the properties of memory devices? Memory policy is the current hammer, and NUMA is the current nail, so the question for today is, how could we categorize NUMA nodes? This doesn't quite capture the reality of the situation, since we don't exactly need categories, but we need orderings. That is, if we want memory on a high-bandwidth device, well, there could be several devices with different bandwidths, some of which are high and some of which are not. The better question, therefore, is, how do we construct a bandwidth ranking of devices?

The first answer that will jump into most people's heads is empiricism. Bandwidth is measurable, so take measurements. This approach is problematic, as the central tenet of empirical approaches is that "incidental" observations are representative of a general phenomenon. This makes testing methodology an important issue, since (e.g.) bandwidth measurements are only meaningful if the testing protocol is representative of the conditions under which the memory will be used, size of access and stride of memory comes into play.

A more compelling answer would be to extend NUMA and/or ACPI to include standardized results (e.g., from well-conducted tests) of various memory metrics for the different NUMA nodes. Such a standard is well outside the scope of this paper, though it is something we would be interested in working on, given enough interest and collaborators.

A final, simple approach would be to require administrators of high-performance computers to maintain a configuration file listing standardized characteristics of the NUMA nodes. This is tractable (in principle) because the population of heterogeneous-memory high-performance computers in the world will likely remain small, and such a configuration only needs to be written at initial setup and following major upgrades. It would then also be up to the system administration to oversee a testing protocol, or else to rely on specifications or similar material from the hardware vendors.

In any case, this section remains speculative because of the simple fact that heterogeneous memory is itself largely speculative at this time. All we can do, therefore, is speculate about what the future may hold.

[1] In fact, there is one trade-off: one can theoretically get higher bandwidth and higher latency by interleaving memory across all NUMA nodes. This is the intention behind the "interleave" kernel policy, but it isn't used much in practice.

2.5 Shared Memory Arenas

The heap memory managers, as implemented at the moment, allow for memory waste due to fragmentation. Although the modern heap managers can be configured to aggressively return free pages to the operating system, the task is further complicated by the more complex data structures the managers use. In addition to the multiple arenas they create, the memory is further split into bins that try to group data of the same size, and extents that belong to the same bin.

In computers with uniform memory, while fragmentation is wasteful, the problem can usually be fixed by installing more DRAM. For high performance memory, like HBM, the size is usually fixed and memory waste becomes a bigger issue.

In the general case of computer use, the memory waste is considered a normal result of a hard optimization problem. Each process has its own resources and the processes generally don't trust each other. In the HPC environment the biggest memory users on a server usually belong to the same job, and while they might run as separate processes (for example, the normal case for MPI applications), they implicitly trust each other. This trust can be used for better memory utilization by creating a shared memory heap manager that can be used across multiple processes.

The proposed API with its arenas fit well with the implementation of a shared memory heap manager. The developer can choose which data to be in private arenas, handled by the local heap manager, and which to be shared with other processes from the same job.

In addition to the benefits of using shared memory heap managers, there are also some drawbacks. Sharing arenas might cause worse memory allocation performance due to lock contention. Data from multiple processes will be interspersed in the same arena, which will require careful handling of data movement. Bugs like buffer overflows can be harder to detect, as the bug could originate from a different process than the one it appears in.

3 IMPLEMENTATION

3.1 Arena Implementation

The arena API is implemented as part of the SICM [6] project. Currently it restricts an arena memory to belong to a single NUMA node. The main functions of the API are:

```
sicm_arena sicm_arena_create(size_t maxsize,
      sicm_device *dev)
int sicm_arena_set_device(sicm_arena sa,
      sicm_device *dev)
void *sicm_arena_alloc(sicm_arena sa, size_t sz)
void *sicm_realloc(void *ptr, size_t sz)
void sicm_free(void *ptr)
sicm_arena sicm_arena_lookup(void *ptr)
```

Upon arena creation, it is assigned to the specified `sicm_device`. If requested, the arena can be moved to another device by using the `sicm_arena_set_device` function. A maximum size of the arena can be specified. If the user tries to allocate more memory than that size, `sicm_arena_alloc` will return `NULL` pointer.

The arenas functionality is implemented as an extension to the jemalloc [5] arenas. The SICM library provides custom hooks to jemalloc that ensure the arena's extents use pages from the appropriate NUMA memory. They also keep track of the extents so the pages can be migrated to another node if requested. The page migration is implemented by using the `mbind` Linux system call. If not all extents for an arena can be migrated, `sicm_arena_set_device` returns an error.

3.2 Integration with Existing Middleware

The interface described in the preceding text is targeted for HPC runtimes and libraries. Advanced programmers could use it directly with in applications we see enabling access to heterogeneous memory for common runtimes such as OpenMP, MPI and Global Arrays, Legion, Charm++, etc. We have initial implementations for MPI, Global Arrays and are evaluating functionality for OpenMP.

Global Arrays[1] is an ideal candidate for integration with our API, having been written with both NUMA and shared memory in mind. What it does not have is the ability to actively choose memory devices or create arenas to use for its allocations. Our API will allow for Global Arrays to do so, which in turn will allow for better performance through better placement of data. Currently, Global Arrays has been modified so that its calls to `shm_open` are redirected to SICM. No selection of memory device is done beyond selecting the first device. The shared memory arenas used for Global Arrays depend on the pthreads support for shared memory mutexes.

Data placement in the context of MPI communications is of paramount importance to achieve high performance: high-performance data transfers over the network is only efficient if the data can be placed correctly in the memory hierarchy and ultimately efficiently accessible by MPI ranks or threads that need it. Unfortunately, the MPI standard and MPI implementations do not provide any mean or interface for the placement of data in complex memory hierarchies. In addition, the community agrees that it is beneficial to provide mechanisms to application developers so that they can express the intent related to the data transfer in order to better select where to store the data once the MPI operation completes.

Based on these constraints, we extended Open MPI to allow users to allocate memory by providing hints while using the existing `MPI_Mem_alloc` function. Our extension consists in the implementation of a new mpool component in the OPAL layer that interfaces with SICM.

Practically, application developers can express hints through the info structure that is passed in when using `MPI_Mem_alloc`. These hints can then be used to allocate memory arenas using SICM. At the moment, such hints includes specifying the need for high bandwidth memories or standard main memory but can easily be extended to other types of memory and others types of requirements from users. Finally, this approach has the huge benefit from not requiring any modifications to the MPI standards or Open MPI interfaces, while still giving control over memory allocation and data placement to users.

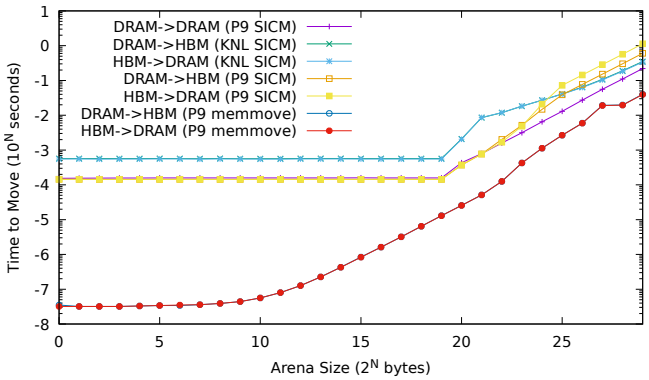


Figure 1: Time To Move Memory Between Devices

4 EXPERIMENTAL STUDY

4.1 Moving Memory Between Devices

Figure 1 shows comparison of the time to move an arena between different devices, depending on the size of the arena. We ran the experiments on two architectures that support heterogeneous memory. The IBM CORAL Power 9 machines have six active NUMA nodes, four with DRAM and two with the high bandwidth memory located on the Nvidia GPU. The Knight's Landing machines have two NUMA nodes, one with DRAM and one with high bandwidth memory. We tested moving arenas from one memory type to another. The DRAM-to-DRAM values show the movement from one DRAM NUMA node to another. Because the KNL machines have only one DRAM node, there is no DRAM-to-DRAM plot for it. Each datapoint represents the average time taken to move an arena with size 2^N bytes. For comparison, we also show the time it takes to move the data with `memmove`.

The results show that data migration is expensive. Moving 0.5 GB of data takes approximately a second. Therefore, arena migration to high performance memory makes sense only if the computational kernel that is using it runs long enough to amortize the cost of migration. Arena migration is much slower than moving the data with `memmove`. The main basis for the slowdown is migration uses the Linux kernel to transport the data to pages of different memory device while preserving the same addresses, while `memmove` doesn't. The initial analysis of the kernel shows that the code for the migration is not optimized for the task, and palpable future work would be to improve the speed of the arena migration.

4.2 VPIC

In order to quantify the usefulness of our API in real world applications, we compared using our API with using `malloc(3)` and `numactl(8)` in VPIC[2], a particle-in-cell simulation code for modeling kinetic plasmas in one, two, or three spatial dimensions.

Figure 2 shows the results of running VPIC with `malloc(3)` running normally, under the influence of `numactl(8) --preferred`, and replaced with our API. The rank count of 64 was chosen to be

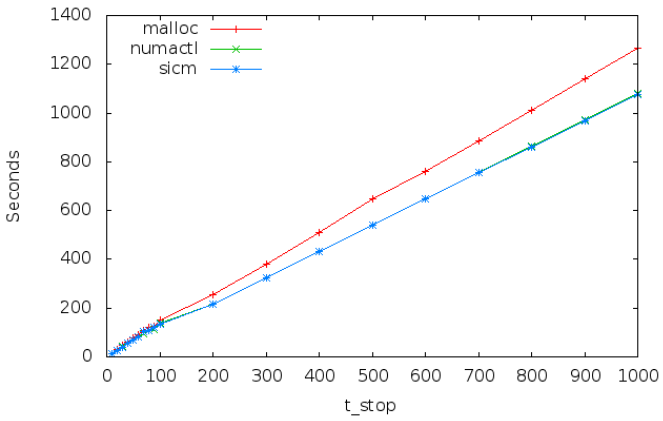


Figure 2: VPIC runtimes with different t_stop values, using `malloc(3)`, `numactl(8)`, and SICM. (64 ranks, nppc=25)

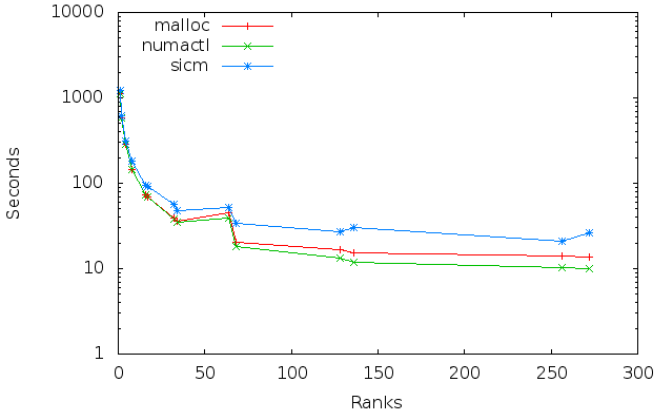


Figure 3: VPIC runtimes with different number Of ranks, using `malloc(3)`, `numactl(8)`, and SICM (with spilling).

small enough to run quickly, while large enough to represent a real problem. The value nppc was set to 25 in order to allow for the entirety of VPIC allocations to reside in high bandwidth memory. Using `malloc(3)` normally results in the longest runtimes of each run of VPIC. Using `numactl(8)` and our API results in lower runtimes. However, our SICM API results in runtimes that are slightly faster than with `numactl(8)`.

Figure 3 shows the results of running VPIC with provided fixed input decks. In these runs, the VPIC allocations were not always able to fit into high bandwidth memory, so a simple spilling function was added into VPIC to use DRAM arenas once high bandwidth memory was exhausted (once DRAM was chosen to be used, high bandwidth memory was not used again during a run). The results show that our API has higher overhead than both malloc(3) and numactl(8) when spilling is required.

5 CONCLUSION

Heterogeneity always presents a serious problem for computer scientists' preference for elegance, and heterogeneous memory is shaping up to be no different. Broadly speaking, we see the problem of heterogeneous memory as consisting of three major parts: controlling how allocations end up on pages, deciding how pages end up under policies, and specifying how policies correspond to actual devices.

The first problem, putting allocations on pages, we addressed through a redefining the meaning of allocator arenas. Under this scheme, we assume that data will move between devices as the program runs, and we give programmers a tool to handle the unintended consequences of page migration. We then argued that this view of arenas also dampens some of the undecidability of the problem of modeling the behavior of computer programs. Finally, we discussed the problems of interpreting memory policies, and presented a few solutions.

We described initial use of the SICM API, and showed preliminary results for micro-benchmarks and an application, VPIC.

Together, we believe this represents a complete "middleware" package for heterogeneous memory, so that we will be poised to tackle its issues once a major heterogeneous-memory supercomputer is built.

As future work, we are planning to extend the API to support arenas on multiple NUMA nodes, as well as a way to specify that an arena can be placed on any available memory, because its data is not in active use at the moment. Also, an asynchronous version of the arena migration will help improving the overall performance of the applications. An important task to look into is improving the performance of the `mbind` Linux implementation and closing the performance gap between `memmove` and the system call.

6 ACKNOWLEDGEMENT

The Simplified Interface to Complex Memory (SICM) is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] 2018. Global Arrays. (2018). https://github.com/GlobalArrays/ga
[2] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 2008. 0.374 Pflop/s Trillion-particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 63, 11 pages. http://dl.acm.org/citation.cfm?id=1413370.1413435
[3] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. 2015. *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies.* Technical Report. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).
[4] George Chrysos. 2014. Intel® Xeon Phi coprocessor-the architecture. *Intel Whitepaper* 176 (2014).
[5] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference, Ottawa, Canada.*
[6] LANL. 2018. SICM – Simplified Interface to Complex Memory. (2018). https://github.com/lanl/SICM
[7] LLNL. 2018. CORAL/Sierra System. (2018). https://asc.llnl.gov/coral-info
[8] NVidia. 2018. NVidia NVLink High-Speed Interconnect. (2018). https://www.nvidia.com/en-us/data-center/nvlink/