Scaling Shared Memory Multiprocessing Applications in Non-cache-coherent Domains

Ho-Ren Chuang
Virginia Tech, USA
horenc@vt.edu

Robert Lyerly
Virginia Tech, USA
rlyerly@vt.edu

Stefan Lankes
RWTH Aachen University, Germany
slankes@eonerc.rwth-aachen.de

Binoy Ravindran
Virginia Tech, USA
binoy@vt.edu

ABSTRACT

Due to the slowdown of Moore's Law, systems designers have begun integrating non-cache-coherent heterogeneous computing elements in order to continue scaling performance. Programming such systems has traditionally been difficult – developers were forced to use programming models that exposed multiple memory regions, requiring developers to manually maintain memory consistency. Previous works proposed distributed shared memory (DSM) as a way to achieve high programmability in such systems. However, past DSM systems were plagued by low-bandwidth networking and utilized complex memory consistency protocols, which limited their adoption. Recently, new networking technologies have begun to change the assumptions about which components are bottlenecks in the system. Additionally, many popular shared-memory programming models utilize memory consistency semantics similar to those proposed for DSM, leading to widespread adoption in mainstream programming.

In this work, we argue that it is time to revive DSM as a means for achieving good programmability and performance on non-cache-coherent systems. We explore optimizing an existing DSM protocol by relaxing memory consistency semantics and exposing new cross-node barrier primitives. We integrate the new mechanisms into an existing OpenMP runtime, allowing developers to leverage cross-node execution without changing a single line of code. When evaluated on an x86 server connected to an ARMv8 server via InfiniBand, the DSM optimizations achieve an average of 11% (up to 33%) improvement versus the baseline DSM implementation.

SYSTOR '20, June 2–4, 2020, Haifa, Israel

CCS CONCEPTS

• Software and its engineering → Software design techniques; **• Computer systems organization → Cloud computing**.

KEYWORDS

System Software, DSM, Heterogeneous Architectures, Shared Memory Programming, InfiniBand

ACM Reference Format:
Ho-Ren Chuang, Robert Lyerly, Stefan Lankes, and Binoy Ravindran. 2020. Scaling Shared Memory Multiprocessing Applications in Non-cache-coherent Domains. In *Proceedings of The 13th ACM International Systems and Storage Conference (SYSTOR '20).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3383669.3398278

1 INTRODUCTION

Data volume is expected to increase by 40% every year of the next decade [25]. This indicates that there is an urgent need for more computational power to deal with such a large data volume. Meanwhile, 42 years of microprocessor trends [39] show we are facing hardware limitations, the slowdown of Moore's Law, the end of Dennard Scaling, and the Dark Silicon effect [17] caused by per-chip power budgets. Two main solutions to continue scaling performance are clusters of systems integrated via high-speed interconnects [1, 28, 31, 33, 34] and specialized hardware [14, 15, 19, 22, 24, 30, 31]. Recent market trends also show that high-end servers with different Instruction Set Architectures (ISAs) such as ARM and PowerPC are pushing into datacenters [4]. For example, Amazon's Web Services (AWS) provide bare metal ARM servers [3]. Using these emerging systems brings power efficiency [4] and performance benefits [26].

Recently there has been increasing interest in coupling systems with non-cache-coherent domains, or simply *domains or* *nodes*, together in datacenters [4]. Systems with non-cache-coherent domains are usually heterogeneous. Examples include mobile Systems on Chip (SoCs) with same-ISA heterogeneous CPUs [31], Xeon/Xeon Phi platforms [5, 24], ARM-based SmartNICs [9] on a different-ISA server, and heterogeneous-ISA servers interconnected via high-speed networking [4, 34]. Coupling diverse architectures together allows users to obtain better performance [5, 19, 26, 34], increase energy efficiency [4, 30, 31, 45], or provide other capabilities such as improved tail latency [21] and security [44]. However, developing applications for such architectures is difficult, as developers traditionally have been forced to use programming models exposing separate physical memory regions, such as the message passing interface (MPI) [20] or compute unified device architecture (CUDA) [41]. Thus the question arises – *how can developers target non-cache-coherent systems with both good programmability and high performance?*

Current networking hardware trends have revived interest in using Distributed Shared Memory (DSM) to program non-cache-coherent systems [1, 4, 5, 19, 30, 31, 34]. Because DSM extends the shared-memory abstraction across discrete computing elements, users can run existing applications across multiple domains as-is. Previous DSM systems, however, were plagued by networking limitations [2]. Additionally, many optimizations proposed to improve DSM scalability relied on weakening the memory consistency models of the DSM [2], making correctly programming such systems challenging. Besides, new networking technologies continue to improve in both latency and throughput – commercially available InfiniBand adapters provide 200 Gbps bandwidth [23]. This has motivated re-investigating DSM as many assumptions about the performance characteristics of such systems are changing.

As shared memory multiprocessors have become mainstream, developers have turned to parallel programming models such as Intel Threading Building Blocks (TBB) [38], Intel Cilk [37], and OpenMP [16] to easily leverage multiple CPU cores for data- and task-parallel computation. These programming models provide a set of primitives to easily spawn multiple threads, distribute parallel work, and synchronize execution. When using these primitives, developers must write their applications in accordance with the programming model's memory consistency semantics. Oftentimes they use weak memory consistency, where updates are only made visible after synchronization operations. This forces developers to write computations that avoid such data races and thus are amenable to parallelization.

Together with the aforementioned trends in networking, we argue it is time to re-investigate using DSM to target systems with multiple non-cache-coherent domains. In particular, we argue that DSM systems can be optimized using weaker consistency semantics and run existing shared-memory (SHM) parallel programs with high performance on these emerging systems. Using DSM allows developers to run existing applications on non-cache-coherent systems as-is, rather than requiring a full re-write to a new programming model like MPI or PGAS languages [10, 12, 33].

In this work, we explore changing the DSM consistency model and implementation to better optimize cross-domain execution. We found several bottlenecks in previous works [4, 34, 40]: significant numbers of invalidation messages, false page sharing, large numbers of read page faults, and large synchronization overheads. To solve these problems, we propose efficient DSM protocol primitives that delay and batch invalidation messages, aggressively prefetch data pages, and perform cross-domain synchronization with low overhead. We follow the existing shared memory multiprocessing programming model to design our Operating System level (OS-level) DSM primitives, which can be easily adopted in parallel programming runtimes. For developers familiar with programming in such relaxed consistency models or for existing legacy applications, our system transparently brings further performance improvement. To prove the applicability of the new primitives, we developed a runtime based on OpenMP [16]. Thus, we seamlessly support cross-domain execution of existing OpenMP applications without changing any line of code. We evaluated a prototype on two heterogeneous-ISA nodes.

The main contributions of this paper are as follows:

- We design an efficient multiple writer DSM protocol, which allows many writers to concurrently write to the same page without coherency;
- Because a multiple writer protocol may not pay off in all scenarios, we describe a heuristic named *smart regions* that selects the best consistency protocol for a given OpenMP work-sharing region;
- We provide a profiling and prefetching mechanism to further reduce the number of read page faults in work-sharing regions;
- We design and implement a low overhead multi-domain synchronization primitive for avoiding redundant page transfers caused by traditional shared memory synchronization in a DSM setting;
- To demonstrate that our design can be easily adopted by SHM parallel programming models, we incorporate the new primitives into an existing OpenMP runtime, allowing existing applications written in OpenMP to benefit from the new mechanisms without changing a single line of code;
- We present a prototype based on Linux and evaluate it on a setup consisting of an Intel Xeon x86 and Cavium

Table 1: Invalidation messages for a SC-based DSM.

Application	Execution Time (s)	# of Invalidation Messages on x86 (ARM)	Avg. Time (us) Per Msg. on x86 (ARM)	Worst-case Total Invalidation Time (s)	Ideal Total Invalidation Time (s)
LUD	180	539162 (631648)	71 (171)	146	2
CG-D	1396	3840819 (3968198)	424 (891)	1396	101.8
BT-C	453	7053388 (7165938)	277 (806)	453	122.1
SP-C	752	11541815 (11390493)	277 (804)	752	199.8

ThunderX ARMv8 interconnected by 56 Gbps InfiniBand, and achieve on average an 11% speedup versus the baseline DSM implementation.

Our complete Linux kernel implementation is available at https://github.com/ssrg-vt/popcorn-kernel/tree/tso-adv.

2 MOTIVATION

In DSM, the main sources of overhead are fetching pages and maintaining page access permissions across nodes. The consistency model dictates when these operations occur. Prior works [4, 5] use sequential consistency (SC), a multiple-reader/single-writer protocol, for cross-node memory consistency. The DSM system relies on the processor's memory management unit (MMU) to detect remote memory accesses, which are handled by the DSM system inside the OS page fault handler. With SC, if a node reads a remote page, the DSM fetches the page data and sets "shared" permissions across the network; other nodes with the page are also set to have shared permissions. If a node writes to a page, the DSM system fetches the page (if not already present) and sets "exclusive" permissions; other copies of the page on other nodes are invalidated. The latency from cross-node page fetches and permissions maintenance is a well-known DSM problem [36].

Invalidation Messages. In an invalidation-based DSM protocol [18], memory consistency is maintained by sending invalidation messages when acquiring exclusive write access. While processing invalidations, both local and remote CPUs do not perform any useful work. In order to understand the impact of invalidation latency on performance, we collected the total number of invalidation messages and measured how long each message took when running several applications on the setup described in Table 2. The results are shown in Table 1. The second column lists the time to run the application across both machines using 16 threads on the Xeon and 96 threads on the Cavium (16 + 96 threads). The third column records the number of invalidation messages generated by each node. The fourth column shows the average time for each invalidation message from each node. Because of the asynchronous and parallel nature of DSM, we estimate the approximate invalidation overhead in the ideal case in which all invalidation requests are handled concurrently (on both nodes) and the worst case in which all invalidation requests are handled serially. The results are shown in the last two columns, respectively. Compared with application execution time, we observe that the worst case invalidation time for applications consumes either most or all of the application execution time. Even in the ideal case, for CG-D, BT-C and SP-C (Section 5) the invalidation overheads consume, on average, 20% of the application's execution time. This demonstrates the impact of cross-node invalidations on performance.

False Page Sharing. With concurrent cross-node execution, multiple threads executing a work-sharing region may write to disjoint sections but not the same variable of the same page. This incurs invalidation and page transfer overheads even though threads do not write to the same addresses on the page. For example, in Figure 1, writes W2, W4, W5, and W6 all write to different variables located on the same page. In this case, the page will bounce between the nodes as they contend for page data and access permissions. The processors will spend time handling redundant remote page faults instead of application computation. This false page sharing impacts each application differently. For example, 4% of write page faults in lower-–upper decomposition (LUD) (Section 5) are caused by threads on separate nodes writing to disjoint sections of a page when using SC.

Cross-Domain Barrier Synchronization. Another problem arises from synchronizing threads across nodes with traditional SHM primitives. On a single node, synchronization primitives in user-space are often implemented using atomic instructions and wait operations like Linux's fast user-space mutex (futex). On multiple nodes, these traditional primitives can be transparently handled by DSM and futex delegation [28]. However, atomic operations to the same variable cause both read and write page faults. These overheads cause cross-node synchronization overheads to balloon as the DSM moves the page repeatedly between different nodes. As shown in Figure 3, barriers using traditional SHM primitives on top of DSM take 846 microseconds, a 20x slowdown versus barriers on the Cavium and even worse for the Xeon. This can have significant impacts on applications – for example, CG-D and hotspot (Section 5) have 18703 and 10800 barriers during their execution, respectively.

3 DESIGN

Our goal is to reduce DSM overhead so as to leverage remote computing power with minimal communication. Two large sources of overhead caused by using SC are invalidation messages and false page sharing (Section 2). Because these

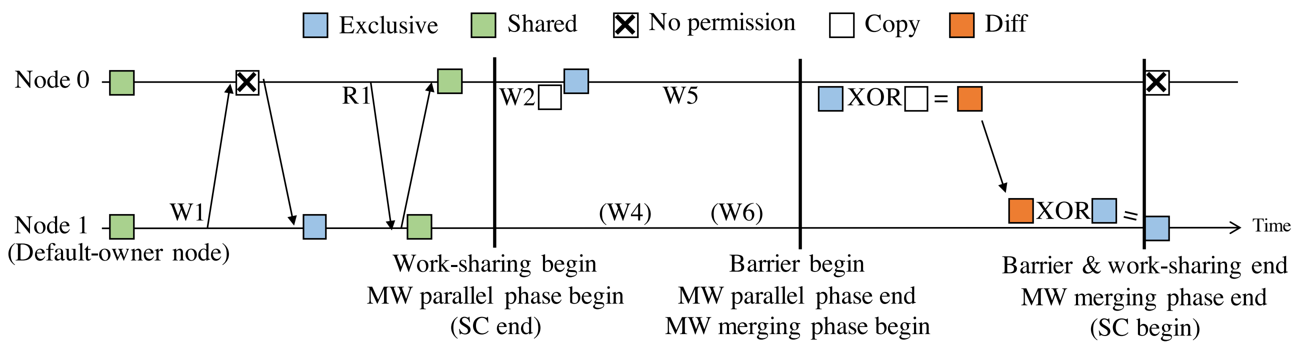

Figure 1: Example of DSM traffic before work-sharing regions (SC) and during work-sharing regions (MW).

are intrinsic to SC, the only way to reduce consistency traffic is to switch to a relaxed model that can delay invalidations and front-load page fetches (Sections 3.1 and 3.2). Also, to reduce redundant communication caused by traditional cross-node synchronization, the DSM includes a new cross-node synchronization primitive (Section 3.4).

3.1 Multiple Writer Protocol

We propose the Multiple Writer (MW) protocol to allow multiple nodes to concurrently write to the same page without cross-node consistency. This allows delaying invalidation messages until the end of a work-sharing region, or simply a region, where all invalidations are exchanged to fix up pages and permissions. For pages written by two or more nodes, the DSM layer detects and propagates updates between nodes. It accomplishes this through Copy-on-Write (COW) – when a node encounters a write page fault, the DSM layer saves a copy of the original version of the page. When propagating writes, the DSM compares the updated page to the original and sends updates to other nodes. This allows updates to be delayed until the end of the work-sharing region and fits directly with parallel programming models.

Upon entering a work-sharing region, the DSM switches from SC to MW and execution enters the *MW parallel phase*. At this point the DSM layer designates one node as the *default-owner* and all other nodes as *non-default-owners*. This facilitates an optimization in propagating updates – rather than an all-to-all exchange for page updates, the default-owner pulls in updates from all other nodes (all-to-one) and performs the merge; subsequent accesses to the page by non-default-owners must reacquire the page and permissions. Because the default-owner merges updates from all other nodes, only non-default-owners perform COW to generate diffs. At the end of the work-sharing region, the DSM layer enters the *MW merging phase*. It begins by exchanging invalidations encountered during the region and determines which pages need merging. Then, the DSM layer generates "differences" (diffs) of the COW pages and sends them to the default-owner. The default-owner merges the diffs, and the DSM system transitions back to SC.

Take Figure 1 for example. It focuses on a single page case. Node0 represents a non-default-owner node and Node1 is a default-owner node. Both nodes are initially outside the work-sharing region and have the page mapped read-only, as per the SC model. When Node1 first attempts write operation W1, it causes a page fault due to the lack of write permissions. During fault handling, Node1 sends an invalidation message to Node0 to revoke Node0's page permissions. Node0 drops the page and permissions and sends an acknowledgement (ACK) message back to Node1. After receiving the ACK, Node1 gains exclusive write access and writes to the page. Next, Node0 performs a read operation R1. Since Node0 does not have any permissions for the page, it sends a message to Node1 asking for the page and shared permissions. Node1 downgrades its exclusive permissions to shared permissions and sends the page data back to Node0. At this point both nodes have the same shared read-only page. Both nodes have the latest version of the page.

When the nodes enter the work-sharing region, the DSM transitions from SC to the MW. Node0 issues W2 generating a write fault. The DSM system first checks whether Node0 is the default-owner node. In this example, since Node0 is not, it makes a copy of the page (COW) and changes the permissions to be exclusive without cross-node coherency. After this initial duplication, writes to the page will not incur page faults for the duration of the MW parallel phase. This saves the latency of sending invalidation messages and permission changes caused by different interleavings of writes W2, W4, W5, or W6.

When the application encounters either an explicit or implicit barrier (end of work-sharing region), the DSM system enters the MW merging phase. First, the DSM batches all delayed invalidation requests into a single message and distributes invalidations across all nodes. Pages written by only one node are simply invalidated on other nodes by the DSM. However, for pages written by multiple nodes (i.e., the page written by W2 and W5 on Node0, and W4 and W6 on Node1), the DSM layer must generate a diff from the copy created during the first write fault. Non-default-owner Node0 creates a diff by applying an exclusive OR (XOR) operation between the original and updated version of the page. The DSM transfers the diff to Node1, which merges the diff with its copy of the page. After all invalidations and merges have been performed, the work-sharing region ends and the DSM transitions back to SC. Note that COW pages are only used during the MW merge phase if nodes write to the same page and therefore necessitate a merge; however, the DSM layer cannot predict whether this will happen and must copy the page, even for pages eventually only updated by a single node.

3.2 Profiling Page Prefetching

Similar to the MW protocol, the primary goal of aggressive page prefetching is to reduce the number of read page faults. Resolving read page faults incurs long latency – the CPU must undergo a mode switch and the DSM system must transfer each page between nodes using the communication layer. Rather than fetching pages on-demand during the middle of computation, the DSM layer prefetches pages in batches at the beginning of a work-sharing region to both reduce interruptions during computation and better utilize the available network bandwidth. Currently, the prefetcher profiles read operation patterns during the initial execution of a work-sharing region and prefetches the same pages before the next invocation of the same region.

3.3 Smart Regions

The MW protocol's benefit does not come for free. For a non-default-owner node, an extra memory copy is required upon the first write to a page. Also, the DSM must record all written pages and copied pages in a list. During the MW merging phase, the DSM iterates over the lists to find conflicts that need merging. If the number of invalidation messages is not significant, the MW overheads may cause a performance hit versus SC. Many compute-intensive applications have repetitive regions exhibiting similar DSM consistency traffic. The DSM system records each region's behaviour, and if the number of invalidation messages is not above a certain threshold, the DSM system smartly skips switching to MW protocol for subsequent executions of the region.

3.4 Cross-Node Barrier Synchronization

One way to provide synchronization across nodes is to delegate futex operations based on DSM [28]. It allows transparently handling existing SHM synchronization primitives. When a thread enters a synchronization barrier, it atomically fetches and increases a shared counter to determine if it is the last arriving thread. During this process, the DSM layer grabs the page containing the counter and invalidates its permissions on other nodes. Subsequent threads entering the synchronization repeat this process until the last thread arrives. Threads waiting on conditions call a wait futex operation to wait in an in-kernel queue. Once awoken, they load a user-space futex variable to synchronize on the counter. The last arriving thread invokes a wake futex operation and increments the user-space futex counter (a write operation). The futex variable is automatically synchronized by the DSM layer, causing redundant page faults on multiple nodes.

To synchronize the in-kernel futex wait queue, futex operations must be delegated to the same node. Before waiting in the queue, futex waits verify that the user-space futex address still matches the wait condition and then sleeps until a futex wake on the same address. While verifying these conditions, the DSM layer locks the pages to prevent any access. This process may cause up to 2 more pages faults due to the lack of read permissions. All in all, when multiple threads on different nodes try to synchronize with each other, the data and ownership of the page will bounce back and forth.

The DSM layer provides a new primitive to synchronize cross-node threads in a single system call. Only one thread has to invoke the system call, which does two things: first, it broadcasts a notification message to other nodes; second, it spins until receiving a response from all other nodes for the same synchronization point. Upon returning from this function, the cross-node synchronization is finished. This causes zero page faults.

4 IMPLEMENTATION

Our system is built on top of Popcorn [4], which implements inter-node thread migration and DSM in kernel space.

We implement the MW DSM protocol (Section 4.1), smart region (Section 4.2), page prefetch (Section 4.3), and cross-node synchronization (Section 4.4) mechanisms at the OS level. We also adopted an OpenMP runtime (Section 4.5) that transparently integrates the aforementioned mechanisms for work-sharing regions. Thus, OpenMP applications targeting simultaneous cross-domain execution in heterogeneous-ISA systems can benefit from these new mechanisms without changing any lines of application code.

4.1 Multiple Writer Protocol

The MW protocol enables multiple writers to concurrently operate on the same page without coherency to amortize networking latency overheads. In our setup (described in Table 2), the Intel Xeon executes the serial phases of the application and is chosen as the default-owner because it has better single-threaded performance than the Cavium ThunderX. Doing so lowers the number of page faults incurred during the serial phase since the default-owner gains exclusive ownership when merging pages.

The kernel maintains per work-sharing region (per-region) data structures to record relevant information (hashkey, pages written, statistics). The MW protocol's implementation can be split into two phases – the MW parallel phase and the MW merging phase.

MW Parallel Phase. When there is a write to a read-only shared page inside a work-sharing region (a write fault), the fault will be trapped by the OS-level page fault handler. During the write fault handling process, the DSM system records the virtual address of the faulting page in a hashmap contained in the per-region data structure; this hashmap is used to construct the delayed invalidation message and detect conflicts during the MW merging phase. Additionally, if it is the default-owner node, it adds write permissions to the corresponding Page Table Entry (PTE) and returns to user space. If it is not, before correcting the PTE, the DSM layer duplicates the page and records the address of the copy in a per-node hashmap indexed by virtual address. After returning to user space, subsequent read or write operations to the page will not incur any further page fault.

MW Merging Phase. Once threads exit a work-sharing region, the DSM enters the MW merging phase. In the merging phase, nodes exchange lists of pages written during the work-sharing region to handle invalidations and detect which pages were written by multiple nodes and therefore need merging. The DSM layer batches invalidation requests up to a tunable threshold (4087) by iterating over the hashmap containing write-faulting pages. Plus, due to OpenMP work-sharing semantics, no two threads will ever write to the same address without synchronization, meaning the merge process will never have to resolve conflicting writes to the same address. We adopt differential logging [29] to both create a differential between the current and copied page, and apply the differential to the same page on the default-owner.

4.2 Smart Regions

The DSM layer can decide whether to use the MW protocol for each work-sharing region. The kernel records execution characteristics from previous invocations of the same work-sharing region to make protocol selection decisions for the next invocation of those regions. If a region is marked as non-beneficial, threads entering the work-sharing region avoid MW meta-data initialization and fall back to SC.

The DSM layer determines whether a region is beneficial at the end of the MW merging phase by recording the number of invalidation messages. If the number of invalidation messages is lower than an *invalidation threshold*, then using the MW protocol does not pay off and it is declared a *probational region*. If a region is determined to be a probational region for consecutive iterations, it is marked as a non-beneficial region and falls back to SC for the rest of execution. Our current invalidation threshold is set to be the core count on its node. The probation threshold is set to 10. We found that some applications have different numbers of invalidation requests for the same work-sharing region. The probation threshold is utilized to gather more profiling information across several invocations of the region to make more accurate decisions regarding whether the MW protocol will be beneficial.

4.3 Profiling Page Prefetching

The in-kernel per-region data structure also records all read faults. The DSM layer uses this record to prefetch those pages in the next execution of the same work-sharing region. The prefetcher relies on the fact that many High Performance Computing (HPC) applications execute the same work-sharing region with the same input/output buffers multiple times, leading to repetitive page access patterns for executions. To identify repeated invocations of a work-sharing region, the DSM layer uniquely identifies regions with a key using the containing function's name, line number, file name, and iteration. This information is supplied to the kernel by the OpenMP runtime (Section 4.5). The key and per-thread hashmap can be used to quickly determine whether the region has been previously executed.

During the first program execution of a work-sharing region, the kernel records read-faulting pages in a hashmap. Upon subsequent program executions of the region, the prefetcher iterates over the faults observed from the previous execution. The prefetcher sends requests as batches, maximizing the number of pages fetched in a single message to better utilize network bandwidth. Upon receiving the response message, the prefetcher maps the pages by fixing permissions and copying the page content to the proper pages. Once pages have been placed across nodes, the threads are released to begin computation.

4.4 Cross-Node Barrier Synchronization

To replace the traditional fetch-and-add and futex synchronization used in the SHM programming model (Section 3.4), we implement a new system call invoked by threads to synchronize across nodes. Each kernel maintains a local barrier counter and a remote barrier counter per remote node. The system call does two things. First, it increases the local barrier counter by 1 and sends a message to the other nodes for increasing remote barrier counter by 1 on remote nodes. Second, the node spins until the remote barrier counter is equal to or larger than the local barrier counter (i.e., until the remote nodes have sent corresponding response messages for

Table 2: Experimental setup

Machine	Intel Xeon E5-2620	Cavium ThunderX
ISA	X86-64	ARMv8
Cores	8 (16HT)	96 (48 * 2 socket)
Clock (Ghz)	2.1 (3.0 boost)	2.0
LLC Cache	L3 - 16MB	L2 - 32MB
RAM (Channels)	32GB (2)	128GB (4)
Interconnection	Mellanox ConnectX-4 56Gbps	

synchronization). Upon completion of spinning, the thread is released back to user-space to continue execution.

4.5 Runtime Support

To utilize the previously described mechanisms, we modified Popcorn's OpenMP runtime (which is derived from GNU's libgomp) to integrate the proposed primitives. Popcorn's OpenMP runtime provides the ability to migrate threads of a team executing a parallel region between nodes to take advantage of remote compute resources. Our modifications to the runtime added/removed 11(+), 3(-) lines of code for replacing futexes with our primitive, 54(+) for the hash function, and 33(+) for our API library.

Regions. Popcorn's OpenMP runtime marks the start of a work-sharing region with calls to __kmpc_dispatch/static_init(). When entering a work-sharing region, the thread performs a system call to set the MW region flag inside the thread's in-kernel descriptor. At this point the DSM layer decides whether to enable prefetching if the region has been previously seen or logging of read faults if it is the first time executing the region.

When a thread hits the work-sharing region exit point (denoted by calls to __kmpc_dispatch/static_fini()), it will unset the MW region flag. At this point the MW merge phase will begin and propagate writes between nodes.

In between an MW begin and end, there may be barriers besides the implicit end-of-work–sharing barrier. In this case, the DSM does not exit the MW region. Threads reaching the barrier will instead invoke the MW merging phase to force a consistent memory view across nodes.

Barriers. OpenMP uses explicit and implicit barriers to synchronize threads. We invoke OS-level synchronization primitives inside the OpenMP runtime in place of the traditional SHM barrier to optimize cross-node synchronization.

5 EVALUATION

We evaluate our DSM system through a series of micro and macro benchmarks to understand how the MW protocol, aggressive page pre-fetching, and new synchronization primitive improve cross-domain performance. Table 2 displays our setup. We envision in the near future heterogeneous domains (likely 2 domains) will appear in datacenters. So, we experiment on a X86-ARM combination integrated via a high speed interconnect to mimic the envisioned future architectures. We implement our prototype using Linux kernel 4.4.137 and OpenMP 4.5 [8]. In our evaluation, we answer the following questions. With zero lines of application code changed,

- How does the new cross-node synchronization primitive improve barrier performance? (Section 5.1)
- How much does the MW protocol improve performance vs. SC? Is the smart region heuristic able to accurately determine when the MW protocol will and will not improve performance? (Section 5.2.3, 5.2.2)
- How much performance improvement does aggressive data page prefetching provide? (Section 5.2.4)

Benchmark Applications. We choose the following benchmarks to represent HPC applications. We select Blackscholes (BLK) from PARSEC (native input) [35]. BT Class C (BT-C), SP-C, EP-C, CG-D from the C + OpenMP version of the NAS Parallel Benchmarks [32] (NPB) from Seoul National University [42]. Due to compiler limitations, CG-D is slightly modified. This does not change the behaviour or the execution time of the application. We additionally select Lava Molecular Dynamics (LavaMD/LAVA), Lower–Upper Decomposition (LUD) and Hotspot2D (HS) from Rodinia [13], which is a benchmark suite for heterogeneous computing. Last, we include an in-house OpenMP version of Kmeans (KM) written by slightly modifying the pthreads version of Kmeans from Phoenix [43]. These benchmarks cover a variety of different computation and memory access patterns.

OpenMP loop-based work-sharing regions distribute parallel work by assigning loop iterations to threads participating in the thread team. Because we use a heterogeneous setup (Table 2) where each server consists of different numbers and different types of cores, we manually skew the ratio of work distributed to each thread based on each core's relative performance. This ratio is determined experimentally per application. As the focus of this work is not to determine the optimal loop iteration scheduler policy, we leave exploring dynamic scheduling policies as future work.

5.1 Micro Benchmarks

Cross-Node Synchronization. To understand the performance improvement gained from the new synchronization primitive, we ran a microbenchmark that executes fifty thousand OpenMP barriers in a loop across both nodes. We created a thread team consisting of 16 threads on the Xeon and 96 threads on the Cavium for a total of 112 threads across the system. The average barrier time for cross-node execution using the original and optimized barrier is shown in Figure 3. For the single-node cases, X86 (16 cores) and ARM (96 cores), synchronization contention overhead caused by OpenMP barriers is only inside the cache hierarchy (as opposed to DSM which causes cross-node page transfers). The remaining two

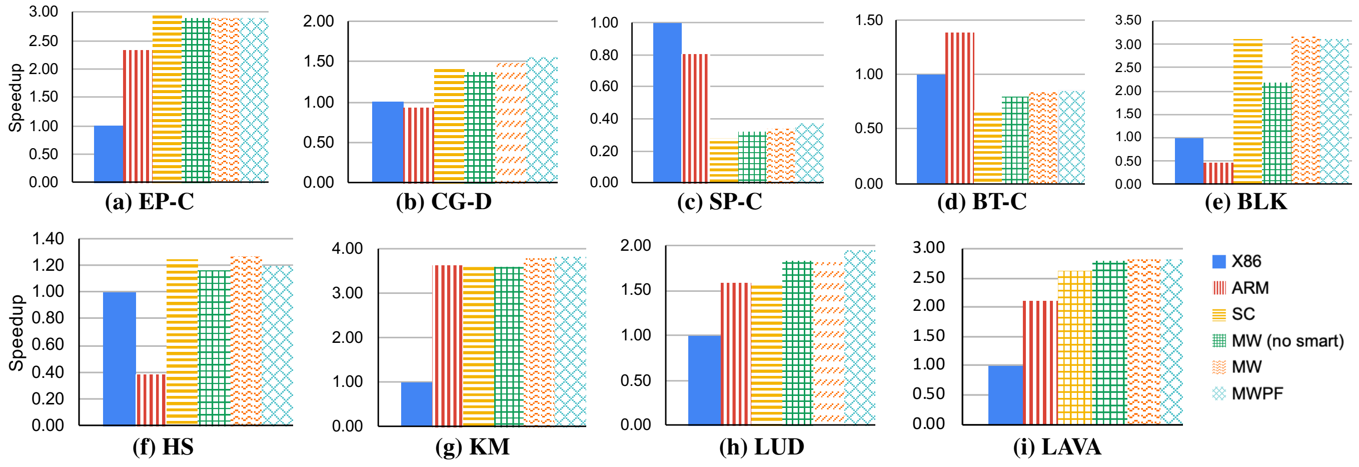

Figure 2: Speedup of benchmarks normalized to single node execution time on the X86 for several system configurations.

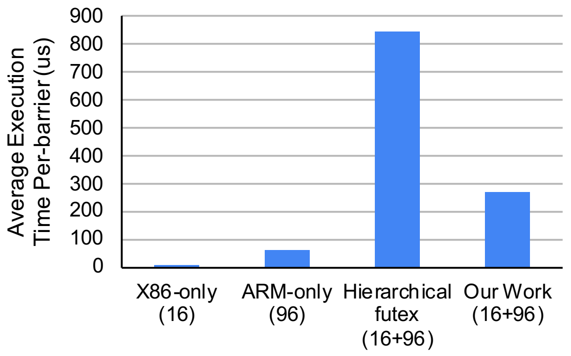

Figure 3: Average execution time per barrier.

bars correspond to the original futex-based barrier as implemented inside the OpenMP runtime and barriers instrumented using in-kernel message passing for synchronization. With the message-passing-based synchronization design, barriers demonstrate a 3.18x speedup compared to solely relying on shared memory and traditional futexes. This shows removing redundant page transfers by using an OS-level cross-node synchronization primitive provides large speedups versus relying entirely on the DSM abstraction while providing a flexible design as well.

5.2 Real Applications

To evaluate performance, we use DEX's [28] kernel-space SC DSM as our baseline, which is built on top of Popcorn [4]. DEX is a working prototype that allows running multithreaded applications across real non-cache-coherent domains. It uses SC, which the gold standard among DSM models because when using a weaker memory model, developing and debugging applications becomes extremely difficult. Indeed, there are many optimized or relaxed consistency models (Section 6.1), but they require code modification and only support heap-allocated objects. Without modification to application code, these DSM systems do not support legacy applications.

5.2.1 ***Experimental Results and Explanation.*** Figure 2 demonstrates the speedup compared to single-node execution on X86 for several system configurations on 9 benchmarks. For each application, the different colored bars on the x-axis show speedups for single-node X86 with 16 threads (always 1), single-node ARM with 96 threads, cross-node execution with 112 threads (16 Xeon + 96 Cavium) with the SC DSM, cross-node with the MW protocol and no smart regions (MW no smart), cross-node with the MW protocol and smart regions (MW), and cross-node with the MW protocol, smart regions and prefetching (MWPF). SC, MW no smart, MW and MWPF all transparently execute SHM applications on two nodes with 16 + 96 threads for parallel regions by leveraging Popcorn's [4]'s thread migration ability. SC does not use any of the optimizations described in Section 3. Conversely, except SC, all adopt the improved cross-node synchronization primitive. We present the result of each application as a smaller plot (Figures 2a~2i).

Our final result shows 7 out of 9 applications perform better than single-node execution when using our DSM design. Among these 7 applications, 6 applications are originally scale-out across nodes when using SC [28] compared with running on a single node. These 9 applications experience improved performance when applying the DSM optimizations included in our design. Compared to SC, MW's average speedup over all 9 benchmarks is 8% and up to 22% for BT-C and SP-C. MWPF is up to 33% faster for SP-C and on average 11% faster than SC. We argue 8%~11% average speedup is very significant with the constraint of not modifying any line of code of existing SHM applications.

5.2.2 ***MW Protocol and Smart Regions****.* Solely using MW without applying the smart region design will, in many cases, make performance worse versus SC – for example, CG-D, BLK, HS, and KM all suffer from using MW without smart regions. This is because the MW protocol is not overhead-free. For those regions with only a few invalidation requests, the MW protocol's overhead will be larger than the benefits brought by the protocol. This is solved by our smart region design. Applying the MW protocol and smart region mechanisms together, all benchmarks except EP, LUD, and LAVA experience performance improvements. The 6 applications that benefit from smart regions have multiple independent work-sharing regions inside the benchmark. Some of the regions are not be able to benefit from the MW protocol. Thus the smart region automatically reverts to SC for these non-benefit regions to avoid excessive MW protocol overheads. Other regions inside these benchmarks do benefit from the MW protocol. For example, BT and SP perform matrix operations, which have a variety of different computations spread across different work-sharing regions and thus some of the regions have very short computation and others have long computation. Another extreme example is BLK, which has about 500 iterations of a single work-sharing region. However, it has very few invalidation requests in each iteration. Hence, the smart region heuristic reverts to the SC DSM. Contrarily, for EP and LAVA, we believe smart regions do not provide significant benefits due to the small number of work-sharing regions (less than 10) in these applications. Because of this, the smart region heuristic does not gather enough information before making a protocol determination. LUD has around a thousand regions. None of invocations of any region are determined as non-beneficial regions. This means the smart region only brings overheads such as collecting data and making decision as none of the regions should fall back to SC. Although using smart regions slightly impairs performance, LUD, LAVA, and EP still run faster using a cross-node configuration than any single-node case. This shows smart regions preserve stable performance results and provide moderately better performance in many cases.

5.2.3 ***MW Protocol with Smart Regions versus SC****.* BT, SP, LUD, LAVA, KM, and CG all benefit from the MW protocol; 98.63%~99.9% of all invalidation messages are batched, allowing applications to enjoy smaller overheads. Among these applications, LUD is the most interesting case as it has parallel regions benefiting from running on a cross-node configuration. However, simply using SC for LUD is not faster than solely running on ARM because the computation benefit afforded by multiple machines is eliminated due to high communication costs. With the MW protocol, invalidation messages in LUD are reduced by 99.77% across nodes. In addition, without using our design, around 4% of the writes cause pages to keep bouncing between nodes. The redesigned DSM boosts LUD's performance by up to 14% compared to using SC. Additionally, when using the MW protocol, BT and SP run 22% faster versus using SC. Conversely, BLK, HS, EP iteratively compute using well-partitioned memory access patterns that do not generate many write faults. While these 3 applications do not benefit from the MW protocol, they do not have obvious performance degradation as well. This shows that even in the worst case of using the MW protocol, it has very low overhead.

In summary, using the MW protocol with smart regions, BT, SP, LUD, LAVA, KM, and CG benefit from disabling MW for only a selected few regions while enabling it for others. The result shows an average speedup over all 9 benchmarks of 8% (with up to 22% for BT-C and SP-C) versus SC. This result proves that the MW protocol indeed can bring better performance by delaying and batching invalidation messages until the end of the region (or barrier) and by solving the false page sharing problem.

5.2.4 ***Profiling Prefetch****.* When using profile-guided prefetching, the results can be categorized into two types. BT, SP, CG, LUD, and KM benefit from the profiling prefetch mechanism. We recorded the number of read page faults reduced in work-sharing regions inside these applications. We discovered 84%~99% of total cross-node page faults were eliminated for these applications after applying the profiling prefetch design. These pages are aggressively prefetched in batches before entering the work-sharing region so as to reduce run-time page fault handling overheads and the latency that the interconnect introduces. HS, however, suffers from 4% slowdown because our current prefetch implementation requires nodes to synchronize for prefetching before entering each region. The overhead of synchronizing is larger than the benefit. The concept of smart regions can also be applied to solve this problem. For, LAVA, BLK, and EP, prefetching does not provide benefit as the applications have only a small number of read faults to begin with. Pre-fetching will not bring much benefit but instead will only add overhead.

6 RELATED WORK

Scaling out SHM applications across nodes is not a new idea. Our unique challenge is how to efficiently leverage existing programming frameworks while transparently scaling out a SHM application on multiple incoherent domains. Compared to previous DSM works [4, 5, 28, 34], our work endeavors to optimize the DSM protocol and implementation to reduce DSM overheads. Our work builds upon a variety of techniques

Table 3: Comparison of related work on leveraging remote computational resources and how much programming effort is required by a developer. oundaries across multiple nodes.

	Traditional & Recent DSM systems [2, 6, 11, 18, 47] [7, 27, 46] [19, 30, 31]	**Dynamic Process Migration [4, 5]**	**Dynamic Thread Migration [28, 34]**
(A) Goal	Programmability	Migrate execution	Distribute execution
(B) Memory model	Shared	On-demand offloading	Shared
(C) Execution replacement	No	Yes	Yes
(D) Relocation unit	-	Process	Thread
(E) Concurrent execution	-	Single-node	Multi-node
(F) Programming effort	High / Low	No	No

and mechanisms [4] such as DSM and thread migration. Table 3 shows a comparison of traditional and recent DSM works, process migration works, and thread migration works including our work. The features of these systems are categorized into different aspects (rows in Table 3) for comparing these works. (A) *Goal* is the primary objective for which these systems were designed. (B) *Memory model* indicates the view of memory in the system. (C) *Execution replacement* means if execution can be dynamically replaced. (D) *Relocation unit* is the minimal migration unit. (E) *Concurrent execution* shows how much of the node(s) is concurrently utilized by threads in a process. (F) *Programming effort* explains how hard it is to use each system to concurrently leverage cross-node computing resources.

6.1 Traditional and Recent DSM Systems

MPI [20] and DSM are two extreme ways to scale out an application. One of our main focuses is to run legacy applications as-is. Thus, MPI is not considered in Table 3. Unlike MPI, DSM systems (Column 1 in Table 3) give the illusion of a single shared memory across non-cache-coherent nodes. Memory is automatically kept coherent across multiple incoherent domains by the DSM system software. Traditional DSM systems automatically transfer data across nodes for applications; developers however have to manually place execution at startup and synchronize/assign work along with execution. Additionally, cross-node execution synchronization mechanisms require developers' involvement.

Recent DSM systems such as K2 [31], Reflex [30], and ADSM [19] adopt DSM to ease programming efforts for these systems. DSM is utilized to enable execution on multiple non-cache-coherence domains by transparently exchanging data. However, in K2 and Reflex, peripheral processor code still executes separately and communicates with remote procedure calls (RPC). Although ADSM significantly relaxes programming efforts for CPU-GPU systems, applications must be rewritten to use their *alloc/free/call/sync* APIs.

The aforementioned DSM systems require refactoring applications to exploit remote computing resources (row F in Table 3). DSM systems do not provide execution replacement, but rather require developers to take distributed execution into consideration while programming. Static execution placement (row C in Table 3) also loses the opportunity to dynamically migrate execution to utilize remote computation.

6.2 Live Process and Thread Migration

While utilizing DSM, [4, 5] (Column 2 in Table 3) provide dynamic process-level migration. By supporting dynamic migration capability, these works dynamically and transparently replace a process on a different node. This gives great improvements in power efficiency [4] and performance [26].

Although Popcorn [4, 5] offers execution migration, it does not fully utilize the benefit of DSM in that these works only demonstrate process migration (D in Table 3) which cannot concurrently leverage multiple nodes' resources (E in Table 3) such as computational power and storage. Additionally, even though researchers [28, 34] (Column 3 in Table 3) proposed a framework to simultaneously run threads of a process on multiple nodes, these works suffer from the same problem of tremendous DSM overhead.

While preserving programmability and considering performance, they neglect cross-node performance, which stresses the DSM (B in Table 3). They use SC DSM leading to excessive communication overheads. We instead provide better DSM primitives to relieve the cross-node overheads that DSM creates. In addition, none of these works provide a solution to efficiently support cross-node synchronization. We propose a design to transparently synchronize execution across nodes with lower overhead.

6.3 Mechanisms for Thread Migrations

Thread Migration and Execution on Different ISA Nodes. Our system exploits OS and runtime extensions such as heterogeneous binaries and dynamic stack transformation [4] to dynamically migrate a thread to other nodes. A thread context consists of live register values and the virtual process address space. The migration process is triggered by a system call. The thread invoking the system call will migrate to a remote node – the kernels cooperate to transfer the thread context to the destination node. The destination kernel then creates a new thread and reconstructs the thread context by using the transferred information. Then, the thread resumes execution by returning back to user space. During execution, in addition to DSM maintaining memory consistency, a stack transformation runtime is required to dynamically transform the stack between ISA-specific formats.

7 CONCLUSION

DSM is a powerful tool to eliminate programming complexity and aggregate remote computational power on different non-cache-coherent domains in the cloud and datacenters. There are many performance bottlenecks caused by existing DSM systems such as too many long-latency invalidation messages, page faults caused by false sharing, and redundant communication from synchronization. These problems are solved by our OS-level MW protocol, aggressive page prefetching and cross-node synchronization. We utilize these design changes for work-sharing regions and develop a runtime to evaluate our design, demonstrating the ease of integrating them into existing parallel runtimes. Our work enables better performance by allowing applications to concurrently execute across non-cache-coherent domains while simultaneously maintaining SHM programmability. Existing SHM applications can transparently leverage our system and scale out. We evaluated our system on a setup composed of an Intel Xeon x86-64 server and a Cavium ThunderX ARMv8 server interconnected with a high-speed network fabric. From the evaluation, we show that versus SC, MW's average speedup over all 9 benchmarks is 8% (MWPF 11%) faster and up to 22% (MWPF 33%) faster with the constraint of not changing a single line of code.

Our complete implementation is available as open-source as part of the Popcorn Linux project: http://popcornlinux.org/

ACKNOWLEDGEMENTS

This work is supported by the US Office of Naval Research (ONR) under grants N00014-16-1-2711, N00014-16-1-2104, and N00014-18-1-2022, and by NAVSEA/NEEC under grant N00174-16-C-0018.

REFERENCES

[1] Jan. 2020. ScaleMP vSMP. https://www.scalemp.com/.

[2] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. TreadMarks: Shared memory computing on networks of workstations. *Computer* 29, 2 (Feb. 1996), 18–28.

[3] Amazon AWS. 2019. Now Available: Bare Metal Arm-Based EC2 Instances. https://tinyurl.com/y6fd7w5n.

[4] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 645–659. https://doi.org/10.1145/3037697.3037738

[5] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 29, 16 pages. https://doi.org/10.1145/2741948.2741962

[6] John K. Bennett, John B. Carter, and Willy Zwaenepoel. 1990. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd PPoPP*. Seattle, WA, USA, 168–176.

[7] Brian N Bershad, Matthew J Zekauskas, and Wayne A Sawdon. 1993. The Midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers*.

[8] OpenMP Architecture Review Board. 2015. OpenMP Application Program Interface v4.5. *Technical Report. https://tinyurl.com/yxzbx5cn* (2015).

[9] Broadcom. Jan. 2020. Stingray™ SmartNIC Adapters and IC. https://tinyurl.com/y6q46rxx.

[10] Francois Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. 2004. Productivity analysis of the UPC language. In *Proceedings of the 18th IPDPS*. Phoenix, AZ, USA.

[11] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and performance of Munin. In *Proceedings of the 13rd SOSP*. Pacific Grove, CA, 152–164.

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 519–538.

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. IEEE, 44–54.

[14] Stephanie Condon. 2017. Intel unveils the Nervana Neural Network Processor. https://tinyurl.com/ydfjwfls.

[15] Ian Cutress. 2017. Qualcomm Launches 48-core Centriq for $1995: Arm Servers for Cloud Native Applications. https://tinyurl.com/yd6obvtl.

[16] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[17] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th ISCA*. San Jose, California, USA, 365–376.

[18] Brett Fleisch and Gerald Popek. 1989. *Mirage: A coherent distributed shared memory design*. Vol. 23. ACM.

[19] Isaac Gelado, John E Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W Hwu. 2010. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the 15th ASPLOS*. New York, NY, 347–358.

[20] William D Gropp, William Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

[21] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. 2017. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings*

of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17). ACM, New York, NY, USA, 625–638. https://doi.org/10.1145/3123939.3123956

[22] Nicole Hemsoth. 2017. Cray ARMs Highest End Supercomputer with ThunderX2. https://tinyurl.com/y95ljwd4.

[23] HPC Advisory Council. Jan. 2018. Introduction to High-Speed InfiniBand Interconnect. https://tinyurl.com/y7xl2df7.

[24] Joel Hruska. 2017. Intel Kills Knights Hill, Will Launch Xeon Phi Architecture for Exascale Computing. https://tinyurl.com/yckk77ar.

[25] IDC. 2014. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. https://tinyurl.com/ya8oasf8.

[26] Mohamed L. Karaoui, Anthony Carno, Robert Lyerly, Sang-Hoon Kim, Pierre Olivier, Changwoo Min, and Binoy Ravindran. 2019. POSTER: Scheduling HPC Workloads on Heterogeneous-ISA Architectures. In *Proceedings of the 24nd PPoPP*. Washington, DC.

[27] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. 1992. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th ISCA*. Queensland, Australia, 13–21.

[28] Sang-Hoon Kim, Ho-Ren Chuang, Robert Lyerly, Pierre Olivier, Changwoo Min, and Binoy Ravindran. 2020. DEX: Scaling Applications Beyond Machine Boundaries. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

[29] Juchang Lee, Kihong Kim, and Sang Kyun Cha. 2001. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 173–182.

[30] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. 2012. Reflex: using low-power processors in smartphones without knowing them. In *Proceedings of the 17th ASPLOS*. London, UK.

[31] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. 2014. K2: A Mobile Operating System for Heterogeneous Coherence Domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 285–300.

[32] NASA Advanced Supercomputing Division. Sep 2017. NAS Parallel Benchmarks. https://tinyurl.com/y47k95cc.

[33] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 ATC*. Santa Clara, CA, 291–305.

[34] Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2017. OS Support for Thread Migration and Distribution in the Fully Heterogeneous Datacenter. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 174–179. https://doi.org/10.1145/3102980.3103009

[35] Princeton University. 2017. The PARSEC Benchmark Suite. http://parsec.cs.princeton.edu.

[36] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. 1996. Distributed shared memory: concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications* 4, 2 (1996), 63–71.

[37] Keith Harold Randall. 1998. *Cilk: Efficient multithreaded computing*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[38] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. "O'Reilly Media, Inc.".

[39] Karl Rupp, M Horovitz, F Labonte, O Shacham, K Olukotun, L Hammond, and C Batten. Feb. 2018. 42 Years of Microprocessor Trend Data. *Figure available on webpage https://tinyurl.com/yyzzm73w* 6 (Feb. 2018).

[40] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia. 2013. A page coherency protocol for Popcorn replicated-kernel operating system. In *Proceedings of the 2013 Many-Core Architecture Research Community Symposium (MARC)*.

[41] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[42] Seoul National University Centers for Manycore Programming. Sep 2017. SNU NPB Suite. https://tinyurl.com/y3jrfrqg.

[43] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. 2011. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*. ACM, 9–16.

[44] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M. Tullsen. 2016. HIPStR: Heterogeneous-ISA program state relocation. In *Proceedings of the 21st ASPLOS*. Atlanta, GA, 727–741.

[45] Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA diversity: design of a heterogeneous-ISA chip multiprocessor. In *Proceedings of the 41st ISCA*. Minneapolis, MN, 121–132.

[46] Yuanyuan Zhou, Liviu Iftode, and Kai Li. 1996. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the 2nd OSDI*. Seattle, WA, 75–88.

[47] Yuanyuan Zhou, Liviu Iftode, Jaswinder Pal Sing, Kai Li, Brian R. Toonen, Ioannis Schoinas, Mark D. Hill, and David A. Wood. 1997. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proceedings of the 6th PPoPP*. Las Vegas, Nevada, USA, 193–205.