CAREER: Application-centric Multidimensional Cooperation to Support Heterogeneous Memory Systems

Memory plays a crucial role in the performance of computing systems. Non-Uniform Memory Access (NUMA) architecture was introduced to address the scalability issue found in multicore systems, since different cores may access their local memory concurrently and independently. However, existing systems were initially designed for the homogeneous UMA (Uniform-Memory Access) architecture, which cannot fully achieve the performance potential promised by NUMA hardware. As applications become increasingly thirsty for data (e.g., Big Data applications), memory systems will become more heterogeneous and sophisticated by the integration of high-density memory such as Non-Volatile Memory (NVM). The increasing sophistication of these heterogeneous memory systems (HMS) make them even harder to exploit efficiently under the existing system design.

This project proposes a paradigm shift towards "application-centric multidimensional cooperation" that supports next-generation heterogeneous memory systems. In this design, the runtime system is in charge of memory management and task scheduling by orchestrating multiple components inside the system. The key basis of the cooperation is memory sharing and reference patterns of applications. The proposed approach is based on the observation that memory sharing patterns of over 90% of objects typically remain the same, even when applications are fed with different inputs. This project proposes the following cooperations: (1) Applications, the user-space memory manager, and the OS memory manager will cooperate to place data objects, in order to reduce remote accesses and enable more optimization opportunities. (2) The runtime system and OS will cooperate to replicate read-only/read-mostly text and data, and schedule tasks intelligently in order to reduce remote accesses, as well as interconnect and memory controller congestion. (3) Applications, the runtime system, and the OS will cooperate to deal with the issues of NVM-DRAM hybrid systems, such as wear-leveling and performance degradation caused by NVM.

Key Words: Multithreading Performance, Heterogenous Memory Systems, NUMA, Application-centric, Multidimensional Cooperation, Cooperative Memory Management, Cooperative Resource Management

Intellectual Merit. This project proposes a paradigm-shift idea, application-centric multidimensional cooperation, that helps achieve the performance potential promised by NUMA-related heterogeneous memory systems. All key components of the system will cooperate together under the orchestration of the runtime system, based on the memory access behavior of applications. This work represents a promising new direction that is different from traditional system design, but without a redesign of existing systems. This project further proposes a suite of dynamic and static tools, runtime systems, and operating system support to enable these performance optimizations, where each will be automated as much as possible.

Broader Impact. This research will provide a significant performance boost for NUMA-related heterogeneous systems, speeding the acceptance of NUMA architecture and NVRAM technology, and thus saving businesses millions of dollars spent on unnecessary hardware. This project includes multiple educational research activities, as well as the development of new courses such as "Software Support for Modern Architecture". The educational impact will include the training of graduate and undergraduate students at the University of Texas at San Antonio (UTSA), contributing to the technology workforce. Since UTSA is a minority-serving institution in South Texas, with a 53% Hispanic undergraduate population, this project will increase geographic and ethnic diversity. This project will provide outreach to under-represented groups, such as Hispanic students at UTSA, and to local com-

munities, through judging for the Science and Engineering Fair, and mentoring students in Youth Code Jam and local high schools.

CAREER: Application-centric Multidimensional Cooperation to Support Heterogeneous Memory Systems

1 Introduction

1.1 Motivation

Memory is a critical component of all computing systems, including servers, desktops, mobiles, and embedded devices. Multiple factors, such as capacity, energy, cost, and performance, must be balanced when building a new computing system [**?**]. This requirement, along with physical limitations [**?**, **?**], drive two trends for heterogenous memory systems (HMS).

- **Employment of Non-Uniform Memory Access (NUMA) architecture:** NUMA can bridge the performance gap between CPU and memory, and adapt to ever-increasing numbers of cores by reducing contention [85], since multiple banks (or nodes) can be accessed concurrently by different cores connected via high-speed inter-connection such as QPI [45] or HyperTransport [25]. NUMA is heterogenous due to the different access latencies between local and remote accesses [80, 42, 64], as well as asymmetric interconnections [64].

- **Incorporation of emerging memory technology, such as Non-Volatile Memory (NVM), into the NUMA architecture**. NVM, such as phase-change memory (PCM) [**?**, **?**, **?**], memresistor [**?**, **?**], and STT-MRAM [**?**], has several game-changing attributes, including high density, low cost, non-volatility, byte-addressability, and low energy consumption, which will be very useful toward accommodating the high capacity requirements of Big Data applications [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. However, the introduction of NVM also brings increased heterogeneity and sophistication of memory systems.

The increasing sophistication of HMS makes efficient exploitation extremely difficult. Given an HMS with various latencies or asymmetric interconnections, determining the optimal placement of data objects and tasks is challenging [17, 64]. Recent studies show that many applications, although they experience no problems when running on a homogeneous architecture (e.g. UMA), cannot reach half of their performance potential on heterogeneous memory systems [56, 28, **?**, **?**]. As future memory systems become increasingly complicated, and future applications grow more thirsty for data, the issue of performance will become increasingly pressing, potentially limiting any continued improvements in computing efficiency.

Although significant research attention has focused on improving the performance of NUMA-related HMS, they share a fundamental issue which prevents them from achieving the performance potential promised by the hardware. This issue is that **insufficient cooperation exists among critical components of the system**: (1) Applications do not provide sufficient information to the underlying runtime and OS, thus it is impossible to manage the placement of objects appropriately; however, proper placement is the key to high performance [**?**, **?**, 60, 41]. (2) There is insufficient cooperation between user-space and the OS memory manager, and between the memory manager and task scheduler, preventing further performance optimizations.

1.2 Summary of Past Accomplishments

As a Ph.D. student, the PI has published papers in premium system conferences, such as SOSP [**?**], OSDI [**?**], OOPSLA [**?**, **?**], and PPoPP [**?**], with a focus on false sharing performance problems and deterministic execution of parallel applications. After coming to UTSA, he has led multiple top publications, such as ASE'17 [**?**], EuroSys'17 [**?**], ICSE'16 [**?**] and CGO'16 [**?**], multiple submissions to SOSP'17 [**?**] and

CCS'17 [**?**], as well as four other ongoing projects. These works focus on synchronization performance issues, synchronization reliabilities, memory vulnerabilities, false sharing problems, security, and educational research in memory management [**?**], which substantially extends the PI's research scope. Except one [**?**], all other works were performed and led by the PI independently, which clearly shows that the PI has the capability to perform the research independently. More updated publications can be found at the PI's website [**?**].

The PI has built a solid background that positions him to successfully pursue the proposed project, including profiling [**?**, **?**, **?**, **?**, **?**], compilers [**?**, **?**], runtime systems [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**], and operating systems [**?**, **?**]. His prior experience will benefit the proposed work in the following ways.

- **Profiling techniques [?, ?, ?, ?]:** Section 3.1 will utilize compiler-based profiling (relating to [**?**]) or binary instrumentation profiling (relating to [**?**]) to identify the memory sharing patterns of applications.
- **Memory management:** Section 3.1 and Section 3.3 describe two novel cooperative memory management methods that will directly benefit from the PI's extensive experience in memory management [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].
- **Source code transformation [?]:** The PI utilized the `Rose` compiler to perform source code transformation to automatically insert lock operations for protecting every shared access. This project will also employ source code transformation to pass sharing patterns to the memory manager.
- **Process-based framework:** The PI implemented a process-based framework that replaces every thread with a process in order to isolate the executions of different threads [**?**, **?**, **?**]. The idea of sharing data across multiple processes will be utilized in the proposed work of Section 3.2.1.
- **Cooperative memory management:** The PI has the experience to orchestrate the memory manager and scheduler together to improve the interactiveness of applications [**?**]. This work inspires the cooperative resource management of this proposal.

1.3 Overview of the Proposed Project

This project focuses on the software-based systematic support for NUMA-related HMS. **We propose a paradigm shift towards the "application-centric multidimensional cooperation" that overcomes the fundamental issue of existing systems**. Figure 1 illustrates the basic ideas of this approach. **Vertically**, applications, the runtime, and the OS cooperate based on the sharing and access behavior of applications. **Horizontally**, the memory manager and task scheduler cooperate inside the user space (see Section 3.2.2). The runtime system is the **central control unit** that guides memory management inside the OS, and controls the scheduling of tasks based on memory sharing and reference patterns of applications (thus, the reason why we call it "application-centric"). This application-centric idea will balance the memory and CPU uses among different nodes for every application, thereby promoting system-level balance. *Note that this radical idea does not require the complete re-design of the existing operating system*. In fact, most OS functionalities remain valid, such as memory management, task scheduling, IO management, and access control, with the OS continuing to be in charge of resource management among different applications.

Memory sharing patterns and memory reference patterns of data objects will be the basis for the proposed strategies. It is driven by the key observation discovered by our preliminary work (Section 2.1): **92% of objects with the same call stack typically have the same memory sharing patterns,**

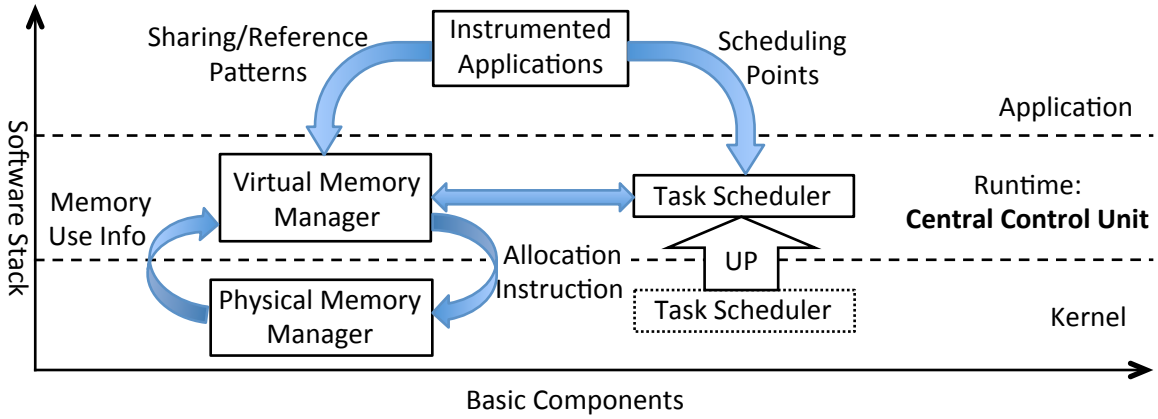


Figure 1: The idea of application-centric multi-dimensional cooperation.

and these patterns are consistent even when programs are fed with different inputs. These memory sharing patterns include private, falsely-shared and shared initially, while the shared pattern can be further divided into read-mostly, producer-consumer, migratory, and randomly-shared. Note that *a misidentified pattern will never cause any correctness issue*, but may lose the potential for performance improvement, as in existing work.

For existing NUMA systems, after obtaining these patterns, the memory manager inside the user space will isolate objects with different types, instruct the OS memory manager to place and replicate them appropriately, and coordinate the task scheduler to access them efficiently (without causing excessive remote accesses). The same idea will be applied directly to the future NVM-DRAM hybrid NUMA systems, but with the additional challenges incurred by the endurance and latency issues associated with NVM.

Overall, three feasible research thrusts are proposed in this project, listed as follows.

The **first thrust** is to enable applications to cooperate with two memory managers in the user space and the kernel level, for existing NUMA HMS. The user space memory manager will partition objects with different types, and instruct page allocations inside the OS, based on memory sharing patterns. Basically, private and falsely-shared types will be handled by this cooperative memory management, while other types will require more sophisticated mechanisms, as described in Section 3.2, to achieve better performance. More details, such as the support for globals and legacy programs, can be seen in Section 3.1.

The **second thrust** is to replicate read-only/read-mostly data and text, and schedule tasks appropriately for migratory and producer-consumer patterns. More descriptions can be seen in Section 3.2.

The **third thrust** is to apply the idea to future NVM-DRAM hybrid NUMA systems. The proposed techniques should handle the challenges brought by the NVM technique. We will also explore the software support for NVM-only architecture in the future. Details are provided in Section 3.3.

2 Preliminary Work

We have performed two preliminary studies in order to confirm the promises of the proposed work. The first study shows two facts: (1) Memory sharing patterns are generally stable, so that they could be utilized as the basis for cooperation. (2) A large number of opportunities exist for optimization. The second study shows the significant performance impact of NUMA-related problems, obtained from

existing work. These problems are currently fixed manually by experts, in a very ad hoc manner, but these corrective strategies could be implemented systematically, as proposed by the project.

2.1 Investigation of Memory Sharing Patterns

Some existing work has focused on the classification of memory references. Some work classifies accesses as random, striding, sequential, or pointer-chasing [72, 69, **?**, **?**]. MemProf classifies remote references of the NUMA architecture into remote uses after allocation, interleaved remote uses, and concurrent remote uses [56]. Others classify references as private, shared, or read-mostly [91], which is similar to our own classification, but misses some of them.

We classify access patterns based on how threads are interleaved, called the "memory sharing pattern" in the remainder of this proposal. At first, we classify memory sharing patterns into **private**, **falsely-shared** and **shared**, where the latter two can be further divided into multiple types, as follows.

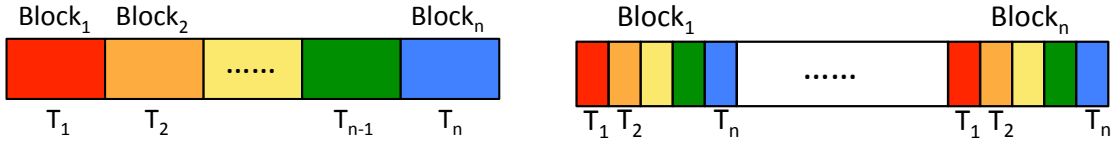
Private pattern: an object is only accessed by one thread. Objects with the private pattern should be placed locally, if possible, in order to prevent remote accesses.

Falsely-shared pattern: the falsely-shared or false sharing pattern is a well-known access pattern responsible for causing performance degradation [**?**, 11, **?**]. The falsely-shared patterns have the internal-object and inter-objects falsely-shared [11, **?**]. Falsely-shared inter-objects are typically caused by the memory allocator, which accidentally places two non-aligned objects into the same cache line or page. The per-thread heap idea introduced by Hoard can be utilized to avoid inter-objects false sharing at the cache-line-level [12]. Typically, most existing work focuses on internal-object false shared [94, **?**, **?**, 68, 33, 66]. To our understanding, internal-object false sharing could be further divided into cache-line-based false sharing and page-based false sharing on the NUMA architecture. The difference between them lies in whether concurrent accesses occur on independent parts of the same cache line or the same page. The UMA architecture only has cache-line-based false sharing, where the PI has extensive experience [**?**, **?**, **?**]. In the NUMA era, cache-line-based false sharing is expected to have an even larger performance impact, since cache invalidations may cause data to be fetched from a remote node unnecessarily [**?**]. Page-based false sharing has been found to be a major source of performance problems for the NUMA architecture [66]. In fact, this type could be further divided into block-wise and interleaved patterns, as shown in Figure 2. These two patterns will require different handling by the memory allocator, as described in Section 3.2.

Shared pattern: Other types will be treated as the shared pattern, which can be further divided into the following types: read-mostly, producer-consumer, migratory, and randomly-shared. For the read-mostly (including read-only) objects, there are multiple readers, but at most one thread writes the data. The examples of this type include text segments, read-only global variables, and read-mostly global variables and heap objects. For the producer-consumer type, one thread writes the data, then another thread consumes (reads/writes) the data. For the migratory type, after one thread accesses the data, it is passed to another thread for access, and then the other [27, 22, 10]. In contrast to the producer-consumer type, there are more than two readers. In addition to these types, an object can be called randomly-shared, where multiple threads may access the data in an interleaved or a random order.

We performed the study of memory sharing patterns on a representative multithreaded benchmark suite–PARSEC [13]–and multiple widely-used multithreaded applications, such as MySQL, memcached, Aget, Pbzip2, and pfscan. We relied on the llvm compiler to instrument memory reads and writes of heap objects, then utilized a runtime library to collect the access information, based on our prior work [**?**]. We will utilize the behavior of the majority of words in an object as the behavior of this

object. We investigated both stability and statistics of memory sharing patterns. The first helps answer whether memory sharing patterns will change when fed with different inputs, while the second helps us to find optimization opportunities.



(a) Block-wise Accesses (b) Interleaved Accesses

Figure 2: Different patterns for internal falsely-shared objects.

Stability of memory sharing patterns: We first confirmed whether the sharing pattern of a call stack (with the same size) will be changed or not, when an application is fed with different sizes and types of inputs. The results are shown in Table 1. In this table, MySQL is evaluated with different types of queries (read-write and read-only), and other applications are fed with different sizes of inputs (large and small). We have these key observations: (1) On average, 78% of callstacks remain the same, even when programs are fed with different inputs. (2) **Among objects with the same callstacks and the same size, 92% actually have the same access pattern, even when they are fed with different inputs**. The observation confirms that *it is possible to pass these stable memory sharing patterns to memory allocation functions*, in order to let the memory allocator place these objects correspondingly.

Application	Callstacks (Large Input)	Callstacks (Small Input)	Same Callstacks	Same Pattern
blackscholes	4	4	0	0
bodytrack	111	109	96	91
canneal	23	21	17	17
facesim	33480	33479	33479	33471
ferret	1879	1879	1852	1564
fluidanimate	8	8	2	2
streamcluster	33	27	16	16
swaptions	23	23	22	22
x264	54	53	18	18
Aget	6	6	4	4
Memcached	18	18	18	16
MySQL	464	401	399	356
Pbzjp2	9	9	8	7
Pfscan	5	5	5	5

Table 1: Callstack information when fed with the large input and the small input. The number of callstacks with the same size are listed in the middle three columns. The "Same Pattern" column lists how many of them have the same memory sharing pattern.

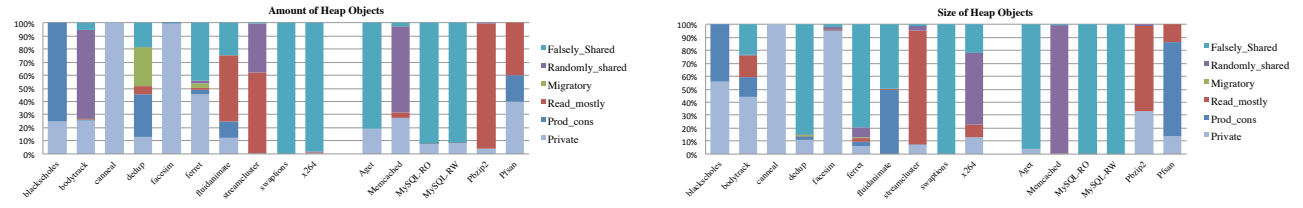


Figure 3: Percentage of access patterns based on the number and total size of heap objects.

Statistics of memory sharing patterns of multithreaded applications: We also investigated the percentage of different sharing types of heap objects in different applications, in order to understand possible optimization opportunities. The detailed results are shown in Figure 3. The left figure shows

Application	Source	Access Pattern	Improving Method	Improvement
AMG2006 [66]	LLNL Sequoia [61]	Falsely-Shared	Block-wise allocation	36%
LULESH [66]	LLNL [51]	Falsely-Shared	Block-wise allocation	7.5%
UMT2013 [66]	LLNL Coral [62]	Falsely-Shared	Block-wise allocation	7%
FaceRec [56]	ALPBench [65]	Read-mostly	Replication & interleaved allocation	26%–41%
Facesim [28]	PARSEC [13]	Prod-Cons&Migratory	Co-location	65%
Streamcluster [56, 28]	PARSEC [13]	Randomly-shared & Read-mostly	Interleaved allocation	37%–161%
Psearchy [56]	Mosbench [19]	Private	Localized allocation	6.5%–8.2%
Apache [56]	Apache [24]	Randomly-shared	Task Co-location	19.7%

Table 2: Examples of performance improvement on applications with different sharing patterns, where all data is collected from the papers listed in "Application" column.

the percentage of different patterns based on the number of objects, while the right figure shows the percentage based on the total size of objects with one type. The total percentage of every application in these figures is 100%. We have the following observations.

- **Potential for replication:** Some applications have a large percentage of read-mostly data, such as pbzip2 and streamcluster, in addition to text segments not covered in the figure. Thus, it is beneficial to replicate these objects to multiple nodes, as in Section 3.2.1.
- **Importance of local accesses:** Some applications — such as blacksholes, facesim, and canneal — have a large portion of data belonging to private data. The memory managers should ensure that they are allocated locally.
- **Potential for task migration:** Some applications have a large portion of objects belonging to the producer-consumer or migratory pattern, such as fluidanimate, blackscholes, and pfscan. For these applications, it is beneficial to migrate threads as described in Section 3.2.2, instead of accessing data remotely.

2.2 Performance Impact

To demonstrate the promises of the proposed ideas, we collected some examples with known performance problems on the NUMA architecture, as well as their corresponding solution, based on existing work [66, 56, 28]. The results are shown in Table 2. Overall, we have the following observations:

- The performance of parallel programs on the NUMA architecture can be substantially improved when remote memory accesses or congestion are reduced, up to 161%. This indicates the importance of the proposed work.
- Different methods, such as block-wise allocation, interleaved allocation, localized allocation, data replication, and task migration, are employed to improve the performance for these examples. These methods match the methods proposed in Section 3.1, Section 3.2.1 and Section 3.2.2, respectively. These sampled programs are currently fixed manually, due to the fact that no systematic solutions exist. Thus, we propose **multiple automatic and systematic methods** to overcome these problems.

3 Proposed Research

We propose three research thrusts in this project: Thrust-I focuses on cooperative memory management that provides the support for Thrust-II and Thrust-III. Thrust-II utilizes the cooperative replication and cooperative scheduling to handle text and data objects with specific shared types, such as read-mostly, migratory, and producer-consumer patterns. Thrust-III talks about the support for future NVM-DRAM hybrid memory systems.

3.1 Thrust-I: Cooperative Memory Management

As described in Section 1.1, existing systems lack sufficient cooperations that prevents them to achieve the performance potential promised by the hardware. In this thrust, the user-space memory manager and the kernel memory manager will cooperate together to physically isolate different types of objects based on memory sharing patterns, and handles objects with private or falsely-shared types.

Memory sharing patterns, as described in Section 2.1, will be the key basis of cooperation. For new programs, these patterns can be provided by programmers by utilizing the new interface, as seen in Figure 4. The `share_pattern` parameter will indicate the type of pattern for this call stack, and the `option` parameter may provide more information for specific types, such as falsely-shared objects.

```
void * nmalloc(size_t size, int share_pattern, int option);
```

Figure 4: New allocation routine.

For existing programs, the patterns can be obtained using the profiling technique. We could run existing programs with multiple representative inputs several times in order to identify the possible sharing patterns. For applications with source code available, we could employ the llvm compiler to instrument the code prior to executions. For applications without source code, we could utilize the binary instrumentation provided by existing tools, such as Pin [**?**], DynamoRIO [**?**], or Valgrind [**?**], in order to obtain memory access information.

But, for any of these mechanisms, we will have a runtime system to collect the memory accesses, then determine the sharing pattern of every call stack, which is similar to our prior work [**?**, **?**]. We plan to record accesses of each word in the shadow memory initially [95], and then identify and integrate the pattern information into its call stack upon object deallocations. When there are multiple sizes from the same call stack, we will save the pattern information based on different sizes at first, then try to summarize the access patterns, if possible.

After the memory sharing pattern of every object has been identified, we may utilize the compiler to instrument code explicitly with the customized API, as in `nmalloc` in Figure 4, when source code is available. If the source code is unavailable, we may utilize a persistent file in which to save the sharing pattern for each call stack, such that the runtime system may use it to check upon every allocation. However, this method may add some runtime overhead.

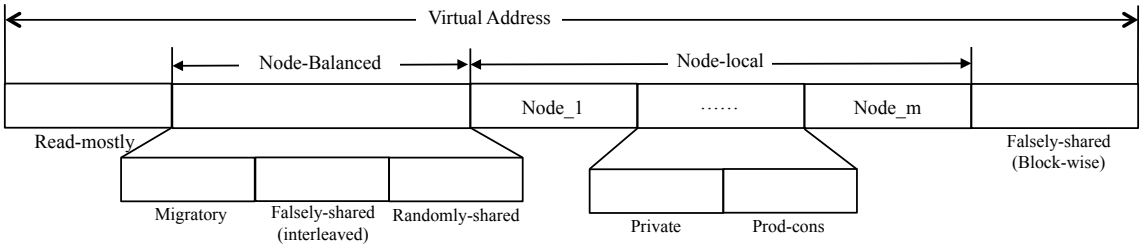


Figure 5: Basic idea of the user-space allocator.

After receiving such sharing patterns, the memory allocator will honor these patterns by isolating objects with different access patterns, as shown in Figure 5. Basically, the virtual address of the heap will be divided into multiple arenas: read-mostly, node-local, node-balanced, and falsely-shared with block-wise accesses. **Read-mostly arenas** obviously will hold objects with read-mostly access patterns, so that we could replicate this region to multiple nodes, as described in Section 3.2.1. **Node-local arenas** will store objects with private patterns and producer-consumer patterns, but they are within different regions. Node-local arenas requires cooperation with the OS memory manager in order to ensure they are allocated in a specific node, in order to reduce remote accesses. For producer-consumer sharing patterns, we may require scheduling assistance in order to guarantee that both producers and consumers will be placed on the same node — a principle known as task co-location — and ensure it is possible to migrate tasks in order to achieve better performance. Section 3.2 presents more details on this, however, the allocator should isolate them at first. Objects inside a **node-balanced arena** will be allocated on different nodes in a balanced manner. The user-space allocator will instruct the OS memory manager to utilize the interleaved-policy for the allocation of physical memory, where `libnuma` APIs will be utilized here [8], or some additional system calls will be added, but without significant changes to the OS.

Falsely-shared objects with block-wise accesses are actually an important source of performance problems on the NUMA architecture [66]. Thus, the memory allocator should handle these objects very carefully. Since the size of the continuous region accessed by one thread can vary among different call stacks or different applications, the memory allocator will be required to collect the possible access size for each thread, and then coordinate with the physical memory manager to decide how much memory should be allocated from each node, as well as how many nodes will be utilized to allocate the memory.

There are some additional implementation problems: (1) How to track the location of every page, which help avoid ownership drifting that may cause a performance cliff? We plan to utilize the shadow memory mechanism to achieve quick lookup on the node location for every page. (2) How to avoid false-sharing problems caused by multiple threads? The per-thread heap organization will be utilized [12], but this may significantly increase the number of arenas when combined with the mechanism listed in Figure 5. We should avoid the memory blowup as much as possible. (3) How to handle global variables with these sharing patterns? We plan to change the loader in order to separate different types.

3.2 Thrust-II: Cooperative Replication and Scheduling

In this thrust, we will replicate read-mostly data and text segments to multiple nodes, and improve the performance of producer-consumer and migratory patterns through cooperative scheduling and memory management.

3.2.1 Cooperative Replication

As observed by existing work, replication helps localize data accesses, distribute workload, and reduce communication and contention [49]. **We propose to utilize a process-based framework to replicate pages**. Since different processes will have separate address spaces and page tables, they could be utilized to map multiple physical pages to the same virtual page. Based on our experience [**?**, **?**, **?**], `clone` system calls and memory-mapped files allow us to share or isolate any part of the address space across multiple processes, but with shared files [**?**, 53].

Performance Concern: We will not create a large number of processes, only equaling the number of nodes in the system. Thus, the performance concern caused by process creation is not an issue here [52,

87, 86, 81]. These processes will be utilized only as memory-holders, and they do not participate in normal scheduling. Due to this reason, the performance concern of cache and TLB flushes upon context switches is no longer valid. Thus, **process-based replication should not significantly increase performance overhead**. However, by replicating pages and page tables, it may increase the memory footprint, which may indirectly affect performance when memory is not sufficient. However, this concern will be reduced with the utilization of NVM technology [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. In order to further reduce performance concerns, only objects with a number of reads exceeding a threshold (obviously-profitable pages) will be replicated, relying on the profiling described previously. Thus, we will only replicate read-only/read-mostly text and data, not including synchronization variables or intensively-written objects.

Correctness Concern: Replicating a page may raise correctness concerns: whether a write on a node can be seen by tasks running on other nodes. Actually, **this problem can be avoided using write-protection, a similar solution to the well-known reader-writer lock problem** [**?**]. For the reader-writer lock problem, only one writer (and no reader) or multiple readers are allowed. Similarly, a page will be write-protected for all processes in order to allow for multiple readers. If a thread in a node writes a page, we will receive a protection fault. Upon faulting, all other nodes will be read-protected immediately (no other readers allowed), but the faulting node will be permitted to write (one writer only). If another node issues a read, then the new version will be copied to that node. Then, that node will be write-protected but readable, and the write permission of the original node will be disabled. However, these operations will be extremely costly. Thus, we will focus on profitable pages only, such as read-only text and data segments, and read-mostly data segments. For read-only text and data, there is no need for these expensive operations. Read-mostly data, if written only once, will not experience any performance problem, as reads generally occur after all writes have concluded.

We could further reduce overhead by using the copy-on-read mechanism, which is similar to "copy-on-write" [**?**]. All pages will be both read-protected and write-protected initially, with a private page created only upon its first access. From that point forward, this page will utilize the mechanism as described above.

Of course, the implementation will include many more engineering-related problems, as in the following examples: changing the ownership for a thread, avoiding multiple copies for the same physical pages, handling the change of page tables, and utilizing large pages for holding read-only segments in order to further reduce TLB thrashing.

3.2.2 Cooperative Scheduling

Figure 1 provides an idea of bringing scheduling into the user-space. However, this does not mean that user-level scheduling will completely replace the OS scheduler. The OS scheduler will still manage different applications within the system, however, the proposed cooperative scheduling will assist the task placement by consulting with the memory manager. We will have the follow two targets.

Initial Scheduling of Threads Based on Memory Requirements and Sharing Patterns: We will utilize memory availability information (obtained from the OS), as well as memory-sharing information across threads, to guide the initial thread assignment. If a memory-hungry thread is placed on a node without sufficient memory, it may cause a large number of swaps unnecessarily [**?**]. Thus, memory-hungry threads should be assigned to nodes with more physical pages available. Also, tasks sharing data will be assigned to different cores in the same node as much as possible, known as **task co-location**, in order to reduce possible interconnect communication, and avoid unnecessary cache misses [**?**]. The latter approach will benefit the task migration described below, by reducing interconnect contention.

Migrating Threads Dynamically Based on Memory Sharing Pattern: Rather than access data remotely, we will migrate tasks for profitable sharing patterns, such as producer-consumer and migratory sharing patterns. Migrating a task can be less expensive than remotely accessing the data directly. But different from existing work [17, **?**], the scheduling points for producer-consumer and migratory patterns are instrumented statically based on profiling, if source code is available (Section 3.1). Incorrect scheduling points may only affect the performance, but not the functionality and correctness of programs, as they are still able to access them remotely. For legacy applications without the source code, we can use the hardware performance counters to detect possible scheduling points, where the PI has abundant experience [**?**]. For machines newer than Haswell, we could sample HITM events which occur when a memory access hits a remote cache line, with a modified state [68, 33].

However, migrating a task is not free, especially when using system calls like `sched_setaffinity()`. Such calls may invoke many operations inside the kernel, such as finding a process by PID, verifying the capability, plus the significant overhead caused by the system call itself (multiple context switches). Thus, we plan to explore the performance overhead of using system calls or utilizing the existing user-space scheduling frameworks to support task migrations [21, 84, 32, 5]. To our understanding, these user-space scheduling frameworks invoke functions for context switches, such as `getcontext` and `setcontext`. These functions will only require to store and load around 256 bytes on Linux, which could be less expensive than the invocation of system calls.

3.3 Thrust-III: Cooperative Support for Future NVM-DRAM Hybrid Systems

New NVM technologies will be integrated into the existing NUMA architecture to accommodate Big Data Applications with high memory requirements [**?**, 6, **?**]. Among different integration methods, hybrid memory — the coupling of NVM with a small portion of DRAM — is thought by experts to be the most-likely method [**?**, **?**, **?**], and is the main focus of this thrust.

Future hybrid NVM-DRAM memory systems may impose additional challenges due to the following facts about NVM. First, it has a much higher write latency and less bandwidth [**?**, **?**, **?**]. Second, it has limited longevity, enduring between 10^6 and 10^9 write cycles, which is significantly lower than that of DRAM (around 10^16 write cycles) [**?**, **?**]. However, **Application-centric multi-dimensional cooperation is still the key to solving problems on such complex heterogeneous memory systems.** We project some additional research, listed as follows.

NVM-aware data structure reorganization: Among existing software, data structures and classes are typically organized based on their functionality and maintainability, without any consideration for the performance and endurance. For instance, some fields of one specific data structure may be accessed intensively, while most of them may be rarely accessed. Based on existing memory allocation policies [**?**, **?**, **?**, **?**, **?**, **?**, **?**], such objects should be placed into DRAM. However, the DRAM may be insufficient to hold all of them simultaneously, due to its scalability issue [**?**]. Some pages will be frequently forced to move back-and-forth between DRAM and NVM/disk, causing the notorious page thrashing issue [29, **?**, **?**, **?**]. Thus, we propose to reorganize data structures using profiling and compiler-based automatic tools, which is similar to existing work, but with different focuses [48]. The profiling tool will trace access patterns of objects and pinpoint possible candidates for reorganization, based on their ratio of access differences. If an object has a dramatically different ratio of access counts across different fields, then this object is possibly a good candidate for reorganization. Then, the compiler-based tools will insert allocations, and change variables upon reference. However, the challenge is that some fields can be referenced directly by address. Thus, we will identify these problems by tracking the number of references on both the old object and corresponding new objects.

Wear-aware and NUMA-aware memory management: As described above, the endurance of NVM is worse than that of the DRAM. No existing memory manager has considered all of these factors together: wear-out, performance, and NUMA. Actually, existing allocators have common practices that may increase the wear-out of NVM. For instance, freed objects are reutilized in Last-In-First-Out (LIFO) order, such as by Linux [63], which may result in some areas being utilized more frequently than others [**?**]. Existing allocators store list pointers at the beginning of a data chunk, which may unnecessarily increase accesses to the area [**?**]. Therefore, Moraru et al. propose to use FIFO for object reutilization, and use a DRAM bitmap to track the states of the objects [**?**]. However, they did not support the NUMA architecture, and cannot guarantee wear-leveling without using randomization, as the PI did for increased security [**?**]. Additionally, objects of different sizes may be reutilized at a different frequency for different applications. We will extend the cooperative management described in Section 3.1 with wear-aware support by designing a dual set of memory management policies: one for NVM, and one for DRAM. Also, the new memory management will instruct the OS memory manager to place objects in NVM or DRAM correspondingly, based on their access patterns, and migrate pages between them, if necessary.

NVM-aware replication and scheduling: Due to the increased heterogeneity caused by NVM, replication and scheduling may pose additional challenges as well. When each node possesses both NVM and DRAM, but with differing ratios of availability, this may affect the placement of tasks. When each node only has one type of memory, the placement of tasks should be obviously different from a DRAM-only architecture. Similarly, replication should involve a decision where to place the replicated pages, NVM or DRAM.

Future explorations: In the future, we will explore the challenge of adopting the NVM-only architecture, which is considered to be more revolutionary and more radical than the hybrid approach [**?**]. Many changes are projected, include paging, page granularity, protection mechanisms, and address spaces, which may indicate an overhauled redesign of the memory manager, and thus all mechanisms put forward in this proposal.

4 Evaluation Plan

We are planning to evaluate these designed systems on NUMA machines with four sockets (available) and eight sockets (will purchase) separately, so that we may determine whether our system can improve scalability. For the NVM-DRAM system, we prefer to evaluate them on the actual hardware. If it is unavailable, we could instead utilize the Quatz emulator to evaluate the performance [**?**]. We will evaluate on different types of multithreaded applications, including the PARSEC benchmark suite [13], many real applications — such as MySQL, Memcached, Firefox, and Apache — and High Performance Computation (HPC) applications [61, 51, 62, 65, 19]. We will evaluate each thrust separately, with each generating at least one separate publication. We will compare our results with those gathered from running on original Linux, and with that of existing work [28]. We expect that our proposed work will greatly improve application performance.

5 Research Schedule

We divide the research into two phases. In the first phase, we will focus on the support of existing NUMA architecture in the first 3.5 years. Then, we will improve our support for systems with NVM. We are planning to hire one graduate student, followed by two more in the 3rd and 4th year, to carry the proposed research activities.

Year 1: Improve the profiling tool for understanding the sharing pattern, develop a compiler-based approach that can transform programs automatically to pass memory sharing patterns, and design the user-space memory allocator to cooperate with the OS memory manager to honor memory sharing patterns.

Year 2: Develop the process-based framework to replicate read-only text segments of applications and libraries, as well as read-mostly heap and globals. We will attempt to reduce unnecessary replications.

Year 3: Develop runtime system support for scheduling tasks intelligently based on memory availability and memory sharing patterns of applications.

Year 4, 5: Reorganize the data structure and develop NVM-aware memory management, replication and scheduling for NVM-DRAM hybrid systems to handle wear-leveling and performance degradation issue. Explore methods to support the NVM-only framework.

6 Related Work

Overall NUMA-related Approaches: There are four types of approaches. The first type migrates threads or physical pages in order to reduce possible contentions, based on online behaviors such as access latency, cache misses, or access patterns [59, 7, 17, 28, 37, 30, 64]. Typically, these approaches do not require the change of programs. However, they have unnecessary online profiling and expensive migration [**?**]. The second class utilizes different policies to manage physical memory inside the OS, such as first-touch [26], next-touch [67, 31], or interleaved policy [54]. But none of these policies will work for all applications or systems with different attributes, e.g. available memory [**?**], access latencies [59, 64], and interconnect contention [17]. The third type changes memory access behavior of programs, in order to reduce possible cache misses and the number of remote accesses [72, 69, **?**]. They are typically difficult to implement correctly. Recently, some researchers starts another direction that requires programmers to annotate programs explicitly [49, 73]. Shoal requests programmers to annotate read/write ratio for array allocations in order to replicate and distribute data correspondingly [49]. Although the idea is intriguing, it only works for arrays in programs written with a special language, but not a systematic solution. TBB-NUMA requires very complicated annotations and only works for work-stealing scheduler [73]. In addition to their specific problems, these existing systems share a common fundamental issue described in Section 1.1.

Profiling tools: There exist a significant number of general-purpose profiling tools [**?**, **?**, **?**] and some are dedicated to NUMA-related performance problems [66, 92, 74, 56]. The proposed profiling is similar to these works, but focuses on a broader scope of memory sharing patterns, based on our classification.

Cooperative memory management: Both general-purpose memory allocators [39, 35, 12, 63] and existing NUMA-aware allocators [77, 88, 71, 50] do not support cooperation between user-space allocators and the OS memory manager, although libnuma and numactl have existed for quite a while [8, 20]. Some existing work focuses on cooperative memory management [34, **?**, **?**, 93, **?**, **?**, 47, 46], but with different aims, such as garbage-collected applications [93, **?**, 46], memory balance among virtual machines [**?**], boosting out-of-core tasks's throughput and response time for interactive tasks [**?**]; Exokernel [34] and Dune [**?**] allow applications to operate on the physical memory management directly. None of them utilize cooperative memory management to improve the performance for NUMA architecture.

Memory Replications: Many existing works can perform memory replications. Hardware-based replications are not practical due to the requirement of specialized hardware [15, 14, 91]; Application-level replications require substantial change to applications, and may not be suitable for general-purpose programs [9, 49]. Existing OS-related approaches only work for special systems that support multiple mappings for the same virtual page [18, 58]. However, they cannot be adapted to modern operating systems, such as Linux. Some works focus on the replication of pages from the page cache [38, 79, 57, 36, 16], which cannot support read-mostly data, another source of optimization opportunities. Carrefour replicates `mm_struct` and PTEs on-demand, when memory read-ratio of a page is above a threshold [28]. However, Carrefour cannot replicate text segments, does not fully utilize existing mechanisms like processes, does not support shared memory across processes or memory-mapped files, and without the cooperation of memory managers.

NVM-related Work: Many existing works have focused on the support of NVM from the architectural point of view [**?**, **?**, **?**, **?**], which is a focus different from ours. Recently, some research concentrates on possible software support for hybrid NVM-DRAM systems [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. Among them, some approaches rank pages based on access frequency and write intensity, and use page migration (very expensive [**?**]) to dynamically re-distribute pages [**?**, **?**]. X-Mem places data structures into appropriate memory types based on the access patterns, such as random, streaming, and pointer chasing [**?**]. However, X-Mem did not classify the data objects as we proposed, and their allocator does not consider wear-leveling.

7 Educational and Outreach Plan

This section discuss the introduce of innovative teaching methods (as the PI have done before), develop new curricula, and increase students involvement, especially minority students at UTSA that has over 50% minority students enrolled [3]. We also project how to outreach to local communities.

7.1 Educational Plan

7.1.1 Educational Research Activities

The PI has been actively involved in educational research activities. He designed a new course project that facilitates the understanding of memory management concepts for undergraduate's Operating Systems courses [89], which will be submitted to the ACM SIGCSE Technical Symposium on Computer Science Education conference (SIGCSE'18) [**?**]. The successful experience, improving students' score by over 20%, confirms Dr. Kolb's conclusion that "learning, change and growth are seen to be facilitated best by an integrated process of data and observations about that experience" [55]. This project is designed completely within user-space, but allows students to practice all important concepts of memory management inside operating systems. It further inspires the following two activities, based on the importance of experience.

Designing a new course to cultivate critical thinking through experience: The PI plans to develop a new course, "Computer Systems Design", to cultivate critical thinking through experience-based learning [70]. This course will target senior students who have completed basic courses such as "Data Structures" and "Systems Programming". The students will be exposed some principles of systems design, and pros and cons of modular design, or cooperative design [**?**]. In the end, they will be requested to develop a semester-long project toward understanding the "Worldwide Trends" of Twits [90]. It will include multiple sub-projects such that students will practice different languages (such as Python and C/C++), different data structures, serialization and parallelization, and different architectures (UMA and NUMA) to solve the same problem. Students are expected to learn that all of these factors may

significantly affect the performance and convenience, with associated pros and cons. **Evaluation plan:** In the end, a survey will be provided to students to evaluate the effectiveness of this approach. Also, some open questions focusing on certain design principles will be tested before and after the class, in order to confirm whether the course helps with critical thinking skills. The corresponding results of applying "experiential learning" to a systems-related course will be submitted to educational conferences such as SIGCSE.

Visualizing the course of concurrency errors and performance degradation: Based on our experience of a course project that studies scheduling algorithms [83], visualization greatly helps students to understand the scheduling of multiple processes at UTSA. We plan to design a visualization tool that helps students to understand the course of concurrency errors, and the possible reason of performance degradation on the NUMA architecture. This tool will combine the PI's research effort in record-and-replay systems [**?**, **?**], and memory management module as described in Section 3.1. To our understanding, there is no such visualization tool. We expect that visualizing program errors and memory placement will help students learn concurrency programming and understand the importance of data placement in NUMA environment. **Evaluation plan:** the effectiveness will be evaluated with survey and examination results, by comparing against students who have not experienced such tool.

7.1.2 Integration of Research and Education

Improving Existing Curricula: The PI plans to improve the "Operating Systems" course for both graduates and undergraduates. Memory management and scheduling are the critical components of these courses. Around 4 or 5 lectures will be added to existing curricula: (1) We will discuss the relationship between user-space memory allocators and the OS memory manager. (2) We will explore the pros and cons of existing memory allocators, and the difficulties of each in supporting NUMA machines. (3) We expect to discuss our research results derived from this proposal.

Offering a topic course on "Software Support for Modern Architectures": The PI designed two new courses: "Kernel Programming" and "Parallel and Distributed Systems". Both received overwhelmingly positive feedback. We plan to design a new course to discuss software support for modern architectures, such as NUMA, neural processors [75, 40, 23], Non-volatile Memory [78, 43], and quantum processors [44]. It will be helpful to prepare our students for modern hardware. This course would be open to senior undergraduate and graduate students. The findings of this proposal will be included in several lectures.

Evaluation Plan: To assess the impact of integration, we will rely on questionnaires, by comparing the answers to questions at the beginning, in the middle of the semester, and at the end. The questions will target students' understanding of corresponding concepts. Finally, we will transform the collected data into meaningful information and actionable items.

7.1.3 Involvement of Students

In the past three years, the PI has involved 7 undergraduate students, 6 master students, and 3 PhD students in the research, through independent study, directed research, thesis, or NSF REU support. These students include two **under-represented Hispanic students**, Zachary J. Rodriguez and Javier Guzman, as well as two military/veteran students, Danny Tsang and Corey Crosser. We plan to involve more students, particularly minority students, in the research through two different methods. First, the PI will include some research tasks of this proposal into certain projects in the "Operating Systems" and "Kernel Programming" courses. An example of such a task is to write a kernel module that identifies the placement of physical pages for an application. Such projects usually pique

students' interest in research activities. Second, the PI will request REU supplements from the 2nd year, in order to involve more undergraduate students through independent studies and REU support. The supplements will be used in conjunction with UTSA's MBRS-RISE program [1], providing research training for talented undergraduates to attract them to research-oriented careers. The PI will recruit new students from who perform well in his instructed courses, and encourage them to pursue CS-related research in the future.

7.2 Outreach Plan

The PI has served as judge for the Annual Texas Science and Engineering Fair, which is typically organized in San Antonio's Convention Center [2]. The PI and his group members have regularly served as mentors for the Youth Code Jam program, "inspiring kids to tell a computer what to do by learning to code" [4]. These activities reach out to K-12 children and their parents in the local communities, helping attract them to the computer science field. In the future, the PI plans to continue these activities and extend the outreach as follows. **First**, we will recruit under-represented students from these activities. **Second**, we plan to involve local high school students into this project, such as Brandeis High School [76]. Researchers have shown that involving high school students into research projects is very effective at attracting them to the computer science field [82]. The PI plans to offer some workshops in high schools to attract more students for summer internships.

8 Broader Impacts

This research will provide a significant performance boost for NUMA-related heterogeneous systems, speeding the acceptance of NUMA architecture and NVRAM technology, and thus saving businesses millions of dollars spending on unnecessary hardware. This project includes multiple educational research activities, and the development of two new courses such as "Software Support for Modern Architecture" and "Computer Systems Design". Educational impact also will include training graduate and undergraduate students at the University of Texas at San Antonio (UTSA), contributing to the technology workforce. Since UTSA is a minority-serving institution in South Texas, with 53% Hispanic undergraduate population, this project will increase the geographic and ethnic diversity. This project will outreach to under-represented groups, such as Hispanic students at UTSA, and to local communities through judging for Science and Engineering Fair, and mentoring students in Youth Code Jam and local high schools.

9 Results from Prior NSF Support

The PI has an active CRII grant and its REU complement, with the title "CRII: SHF: EVID: Evidence-Assisted Detection and Elimination of Memory Errors in Single and Multi-threaded Programs" (Award Number: CCF-1566154; Amount: 190,731; Duration: 03/01/16 to 02/28/18).

Intellectual Merit: This grant proposes a novel record-and-replay framework that can identically reproduce the execution of programs, and two evidence-assisted approaches to identify and tolerate memory vulnerabilities. The grant has already led to **three publications at top venues**, including ICSE'16 [**?**], EuroSys'17 [**?**],and ASE'17 [**?**], and two submissions to SOSP [**?**] and CCS [**?**], and four other ongoing projects includes one educational research activity [**?**].

Broader Impacts: The grant has supported two Ph.D. students and four undergraduate students at UTSA. The grant has led to four patents in application and one graduated master student.

References

[1] Rise at the university of texas at san antonio. `http://www.utsa.edu/mbrs/`.

[2] Texas science and engineering fair. `https://www.txsef.org/`.

[3] Utsa strategic plan. `www.utsa.edu/tierone/strategy/2016/5.html`.

[4] Youth code jam — learn to code in san antonio. `www.youthcodejam.org/`, 2016.

[5] Ralf S. Engelschall. Gnu pth - the gnu portable threads. `http://www.gnu.org/software/pth/`.

[6] N. Agarwal and T. F. Wenisch. Thermostat: Keeping dram hot and nvram cool. *Memory*, 4:6, 2017.

[7] J. Antony, P. P. Janes, and A. P. Rendell. Exploring thread and memory placement on numa architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *Proceedings of the 13th International Conference on High Performance Computing*, HiPC'06, pages 338–352, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] A. K. at SUSE LINUX. "a numa api for linux". `http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf`, 2012.

[9] H. E. Bal, M. F. Kaashoek, A. S. Tanenbaum, and J. Jansen. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency: Practice and Experience*, 4(5):337–355, 1992.

[10] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterisation of splash-2 and parsec. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 86–97, Washington, DC, USA, 2009. IEEE Computer Society.

[11] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. *Munin: Distributed shared memory based on type-specific memory coherence*, volume 25. ACM, 1990.

[12] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 117–128, New York, NY, USA, 2000. ACM Press.

[13] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[14] R. Bisiani and M. Ravishankar. Plus: A distributed shared-memory system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 115–124, New York, NY, USA, 1990. ACM.

[15] D. L. Black, A. Gupta, and W. D. Weber. Competitive management of distributed shared memory. In *Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*, pages 184–190, Feb 1989.

[16] J. Blackwood. An overview of kernel text page replication in redhawk linux6.3. `https://www.concurrent.com/wp-content/uploads/2015/04/kernel-page-replication.pdf`, 2012.

[17] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[18] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 19–31, New York, NY, USA, 1989. ACM.

[19] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[20] F. Brandenburger and C. Wickman. "libnuma". http://oss.sgi.com/projects/libnuma/.

[21] Brian Watling and Michael Ploujnikov. A user space threading library supporting multi-core systems. https://github.com/brianwatling/libfiber.

[22] S. Byna, X.-H. Sun, W. Gropp, and R. Thakur. Predicting memory-access cost based on data-access patterns. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pages 327–336, Sept 2004.

[23] T.-H. Chan, K. Jia, S. Gao, J. Lu, Z. Zeng, and Y. Ma. Pcanet: A simple deep learning baseline for image classification? *IEEE Transactions on Image Processing*, 24(12):5017–5032, 2015.

[24] A. Community. The apache http server project. https://httpd.apache.org/, 2016.

[25] H. Consortium. Hypertransport. https://en.wikipedia.org/wiki/HyperTransport.

[26] D. Cortesi. Origin 2000 and onyx2 performance tuning and optimization guide. http://techpubs.sgi.com/library/dynaweb_docs/0650/SGI_Developer/books/OrOn2_PfTune/sgi_html/ch08.html, 1998.

[27] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 98–108, New York, NY, USA, 1993. ACM.

[28] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM.

[29] P. J. Denning. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 915–922, New York, NY, USA, 1968. ACM.

[30] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß. kmaf: Automatic kernel-level management of thread and data affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 277–288, New York, NY, USA, 2014. ACM.

[31] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach. Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 125–137, New York, NY, USA, 2016. ACM.

[32] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, Nov. 2006.

[33] A. Eizenberg, S. Hu, G. Pokam, and J. Devietti. Remix: Online detection and repair of cache contention for the jvm. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 251–265, New York, NY, USA, 2016. ACM.

[34] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[35] J. Evans. Scalable memory allocation using jemalloc. https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919/.

[36] L. P. Fredrik Teschke. Linux numa evolution, 2016.

[37] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[38] P. Gaughen. Numa status, 2002.

[39] S. Ghemawat and P. Menage. "tcmalloc : Thread-caching malloc". `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[40] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[41] M. Hirzel. Data layouts for object-oriented programs. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 265–276, New York, NY, USA, 2007. ACM.

[42] C. Hollowell, C. Caramarcu, W. Strecker-Kellogg, A. Wong, and A. Zaytsev. The effect of numa tunings on cpu performance. *Journal of Physics: Conference Series*, 664(9):092010, 2015.

[43] L. L.-C. Hsu, J. A. Mandelman, and F. Assaderaghi. Process for forming a memory structure that includes nvram, dram, and/or sram memory structures on one substrate and process for forming a new nvram cell structure, May 15 2001. US Patent 6,232,173.

[44] IBM. Ibm quantum experience. `http://www.research.ibm.com/quantum/`.

[45] Intel. An introduction to the intel quickpath interconnect.

[46] M. R. Jantz, F. J. Robinson, P. A. Kulkarni, and K. A. Doshi. Cross-layer memory management for managed language applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 488–504, New York, NY, USA, 2015. ACM.

[47] M. R. Jantz, C. Strickland, K. Kumar, M. Dimitrov, and K. A. Doshi. A framework for application guidance in virtual memory systems. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 155–166, New York, NY, USA, 2013. ACM.

[48] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 179–188, New York, NY, USA, 1995. ACM.

[49] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. Shoal: Smart allocation and replication of memory for parallel programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 263–276, Berkeley, CA, USA, 2015. USENIX Association.

[50] P. Kaminski. "numa aware heap memory manager". `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/NUMA_aware_heap_memory_manager_article_final.pdf`, 2012.

[51] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[52] J. Kepecs. Lightweight processes for unix implementation and applications. In *USENIX Association Conference Proceedings*, pages 299–308, 1985.

[53] M. Kerrisk. Clone(2) linux programmer's manual. `http://man7.org/linux/man-pages/man2/clone.2.html`.

[54] A. Kleen. A numa api for linux. `http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf`.

[55] D. Kolb. *Experiential learning: experience as the source of learning and development*. Prentice Hall, Englewood Cliffs, NJ, 1984.

[56] R. Lachaize, B. Lepers, and V. Quéma. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[57] P. Ladd. Numa memory architectures and the linux memory system, 2016.

[58] R. P. Larowe, Jr. and C. Schlatter Ellis. Experimental comparison of memory management policies for numa multiprocessors. *ACM Trans. Comput. Syst.*, 9(4):319–363, Nov. 1991.

[59] R. P. LaRowe, Jr., J. T. Wilkes, and C. S. Ellis. Exploiting operating system support for dynamic page placement on a numa shared memory multiprocessor. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 122–132, New York, NY, USA, 1991. ACM.

[60] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 129–142, New York, NY, USA, 2005. ACM.

[61] Lawrence Livermore National Laboratory. ASC Sequoia Benchmark Codes. `https://asc.llnl.gov/sequoia/benchmarks/`, 2013.

[62] Lawrence Livermore National Laboratory. LLNL Coral Benchmarks codes. `https://asc.llnl.gov/CORAL-benchmarks/`, 2013.

[63] D. Lea. The GNU C library. `http://www.gnu.org/software/libc/libc.html`.

[64] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 277–289, Berkeley, CA, USA, 2015. USENIX Association.

[65] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 34–45, Oct 2005.

[66] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on numa architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM.

[67] H. Löf and S. Holmgren. Affinity-on-next-touch: Increasing the performance of an industrial pde solver on a cc-numa system. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 387–392, New York, NY, USA, 2005. ACM.

[68] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. Laser: Light, accurate sharing detection and repair. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 261–273. IEEE, 2016.

[69] Q. Luo, C. Liu, C. Kong, and Y. Cai. *MAP-numa: Access Patterns Used to Characterize the NUMA Memory Access Optimization Techniques and Algorithms*, pages 208–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[70] C. Mainemelis, R. E. Boyatzis, and D. A. Kolb. Learning styles and adaptive flexibility testing experiential learning theory. *Management learning*, 33(1):5–33, 2002.

[71] Z. Majo and T. R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 11–20, New York, NY, USA, 2011. ACM.

[72] Z. Majo and T. R. Gross. Matching memory access patterns and data placement for numa systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 230–241, New York, NY, USA, 2012. ACM.

[73] Z. Majo and T. R. Gross. A library for portable and composable data locality optimizations for numa systems. *ACM Trans. Parallel Comput.*, 3(4):20:1–20:32, Mar. 2017.

[74] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 87–96, March 2010.

[75] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4. IEEE, 2011.

[76] NISD. Brandeis high school. `https://nisd.net/brandeis/`.

[77] T. Ogasawara. Numa-aware memory manager with dominant-thread-based copying gc. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 377–390, New York, NY, USA, 2009. ACM.

[78] S. Ogura. Low voltage eeprom/nvram transistors and making method, July 14 1998. US Patent 5,780,341.

[79] osgx. Shared library bottleneck on numa machine, 2012.

[80] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, A. Bhatele, P. O. Navaux, J.-F. Méhaut, and L. V. Kalé. Improving parallel system performance with a numa-aware load balancer. Technical report, 2011.

[81] M. L. Powell, S. R. Kleiman, S. Barton, D. Shan, D. Stein, and M. Weeks. Sunos multi-thread architecture. In *The SPARC Technical Papers*, pages 339–372. Springer, 1991.

[82] A. Repenning, D. C. Webb, K. H. Koh, H. Nickerson, S. B. Miller, C. Brand, I. H. M. Horses, A. Basawapatna, F. Gluck, R. Grover, K. Gutierrez, and N. Repenning. Scalable game design: A strategy to bring systemic computer science education to schools through game design and simulation creation. *Trans. Comput. Educ.*, 15(2):11:1–11:31, Apr. 2015.

[83] S. Robbins and K. A. Robbins. Empirical exploration in undergraduate operating systems. In *Proc. 30th SIGCSE Technical Symposium on Computer Science Education*, pages 311–315, 1999.

[84] Russ Cox. Libtask: a coroutine library for c and unix. `https://swtch.com/libtask/`.

[85] N. Singer. More chip cores can mean slower supercomputing, sandia simulation shows. *Sandia National Laboratories News Release*, 2009.

[86] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX Summer*, 1992.

[87] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. Mach threads and the unix kernel: The battle for control. In *in Proceedings of the USENIX Summer Conference, USENIX Association*, pages 185–197, 1987.

[88] M. M. Tikir and J. K. Hollingsworth. Numa-aware java heaps for server applications. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 108b–108b, April 2005.

[89] D. Z. Tongping Liu. Memory allocator project, 2016.

[90] Twitter. Worldwide trends. `https://twitter.com/hashtag/worldwide?src=hash`.

[91] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on cc-numa compute servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 279–289, New York, NY, USA, 1996. ACM.

[92] R. Yang, J. Antony, A. Rendell, D. Robson, and P. Strazdins. Profiling directed numa optimization on linux systems: A case study of the gaussian computational chemistry code. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 1046–1057, May 2011.

[93] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 103–116, Berkeley, CA, USA, 2006. USENIX Association.

[94] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *The International Conference on Virtual Execution Environments*, Newport Beach, CA, Mar 2011.

[95] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *The International Conference on Virtual Execution Environments*, Newport Beach, CA, Mar 2011.