AUTOMATED TECHNIQUES FOR ENHANCING DEVELOPER PRODUCTIVITY ON DISAGGREGATED SOFTWARE STACKS

BY

BRIAN RICHARD TAURO

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved [illegible]
Adviser

Chicago, Illinois
December 2024

ACKNOWLEDGMENT

I begin by expressing my deepest gratitude to my Lord Jesus Christ for His unwavering grace and guidance throughout this journey. This achievement would not have been possible without His strength. I am profoundly thankful to my newborn, Casey Faith Tauro, whose perfect timing and boundless joy have been a constant source of inspiration. Your arrival, shortly after my PhD comprehensive exam, was nothing short of divine. To my beloved wife, Akshaya Brian Tauro, your steadfast love and unwavering support has made this journey special. I am grateful for your sacrifices and for walking alongside me every step of the way.

I owe an immeasurable debt of gratitude to my parents, Richard and Irene Tauro, who have given so much, even at the expense of their own health, to see me succeed. Your selflessness and unwavering faith in me have been my greatest source of motivation. I am also deeply thankful to my church family for their prayers and steadfast encouragement, which kept me grounded and uplifted throughout this journey. A special thank you to Pastor Robert and Pramilla Sathuri for their extraordinary kindness in caring for my wife and child during critical moments. Your generosity allowed me to focus on completing my work with peace of mind.

I am profoundly grateful to my advisor, Dr. Kyle Hale, for his trust, patience, and exceptional guidance. His mentorship has been instrumental in shaping my academic journey and helping me achieve my career goals. Your unwavering support and wisdom have been invaluable, and I deeply appreciate all you have done to guide me through this process. I also extend my thanks to Dr. Peter Dinda for his thoughtful feedback and for being available to brainstorm ideas that helped refine various aspects of my research. A heartfelt thanks to Dr. Simone Campanoni for introducing me to compilers, a pivotal skill that shaped the latter half of my research. I am equally grateful to Dr. Rolf Riesen and Dr. Balazs Gerofi for providing industry experience and helping me deepen my understanding of multi-OS systems.

To my collaborators, I deeply appreciate your partnership and dedication to pushing the boundaries of what we could achieve together. Each of you has played an integral role in reaching this milestone. I am especially thankful to Dr. Alexandru Iulian Orhean for his friendship and unwavering encouragement, which helped me stay optimistic and focused. I also extend my gratitude to my labmates—Amal Rizvi, Conghao Liu, Nick Wanninger, Nanda Velugoti and Ian Dougherty—for the many fun and productive brainstorming sessions that enriched my research experience. Finally, I thank Dr. Brian Suchy for generously sharing his time and expertise on compiler runtime techniques.

Lastly, I am sincerely grateful to my committee for their flexibility and support, which enabled me to complete my PhD on time.

AUTHORSHIP STATEMENT

I, Brian Richard Tauro, attest that the work presented in this thesis is substantially my own. In accordance with the disciplinary norms of Computer Science (as outlined in the IIT Faculty Handbook, Appendix S), I would like to acknowledge the following collaborations that contributed to the thesis:

My advisor, Dr. Kyle Hale, contributed to the experimental design, guided data interpretation, and provided clarity on the concepts throughout all the contributions detailed in this thesis.

Conghao Liu and Dr. Kyle Hale contributed to the analytical speedup models and papers for predicting speedup in OS-disaggregated systems, which are included in Chapter 3 of this thesis. Dr. Brian Suchy, Dr. Peter Dinda, Dr. Simone Campanoni, and Dr. Kyle Hale contributed to the TrackFM paper, which is presented in Chapter 4 of this thesis. Ian Dougherty and Dr. Kyle Hale contributed to the CARDS research, which is detailed in Chapter 5 of this thesis. Refer to Chapter 7 for additional details.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The rapid advances in specialized hardware, capabilities including FPGAs, TPUs, microsecond network interconnects, RDMA, GPUs, and other technologies have posed significant challenges for the traditional monolithic client/server model in efficiently managing and scaling resources, both within industry and HPC systems. This challenge has instigated the emergence of a new paradigm known as resource disaggregation, which tackles the limitations of the monolithic client-server paradigm by enabling independent scaling of components such as compute, memory, and storage. The disaggregation of resources can be managed entirely through hardware or software solutions. Resource disaggregation enhances resource utilization by fine grained scheduling of hardware resources to support dynamic workloads efficiently and empowers data centers to effectively address varying computational demands, optimize performance, and curtail operational costs, marking a pivotal evolution in modern computational infrastructure.

However, software based disaggregation, whether over a single server or across multiple servers, has placed an increased burden on developers. They are compelled to continuously adapt to new software stacks and migrate applications accordingly. In some instances, despite the considerable porting efforts, the outcomes may not justify the investment. Unfortunately, much of the existing research fails to adequately address the engineering investment challenge, instead prioritizing new software stacks primarily for performance gains, albeit often at the expense of developer productivity.

This thesis focuses on improving developer productivity in software disaggregated environments and advocates that unless a developer has evidence they should not have to switch to a new software system or OS environment. Even when there is benefit for doing so, software tools should be able to prioritize compatibility by leveraging advancements in low-level system software stacks like the operating system and compilers.

We found initial evidence on ways to improve developer productivity in software disaggregated systems by exploring analytical models backed with emulators to place bounds on application performance. Our speedup models equip developers with a tool to decide whether an application would benefit from resource disaggregation before actually trying to use such a system. While analytical models help developers gain insight before adapting to a new environment, developers may still have to port their applications to achieve high performance. We found out through TrackFM that compilers can enable automatic porting of applications with high performance, thereby improving developer productivity on memory disaggregated systems. One of the limitations of TrackFM was that the runtime memory policies had to be determined at static time for applications, which can lead to performance overheads for certain applications. We overcome this problem by building CARDS, a system that determines far memory policies proactively on software disaggregated systems by combining compiler and runtime information for each data structure within an application automatically. CARDS provides developers with a new alternative that determines far memory polices dynamically instead of using a complex profiling based system to improve policies. CARDS is built on top of TrackFM and overcomes the limitations of static compilers by codesigning compiler analysis with the runtime which enables informed policy decisions at data structure granularity.

The co-design of modern compiler analysis with runtime systems opens a unique opportunity to create tools that enhance developer productivity within resource-disaggregated architectures. We also envision that such codesign can be extremely helpful in emulation of experimental hardware architectures to provide insights quickly without any application porting effort. Leveraging my expertise in low-level system software, my thesis aims to advocate for the integration of automated tools in software disaggregated systems to prioritize developer productivity in datacenter environments.

CHAPTER 1
INTRODUCTION

The rapid advancements in specialized hardware have highlighted the limitations of the traditional monolithic datacenter server model in efficiently managing resources. This has led to growing interest in resource disaggregation, which aims to overcome the challenges of the monolithic server model by decoupling resources across servers within a datacenter. In other words, resource disaggregation separates CPU, storage, and memory resources, traditionally tied to a single server, into distinct nodes that are managed independently. This separation can be managed entirely through hardware or software solutions. Hardware-based disaggregation examples include Intel rack scale disaggregation [2], Huawei DC [3], dReDBox project [4], Firebox [5], and memory blades [6]. On the other hand, software-based disaggregation examples include, but are not limited to, Fastswap, Infiniswap, DSM, Intel mOS, IHK/McKernel and TMO [7, 8, 9, 10, 11, 12].

My thesis focuses on software-based disaggregation, which uses software abstractions such as operating systems (OSes), programming languages and library support to manage hardware resources efficiently. For instance, in a kernel based memory disaggregated system, the swap backend within the operating system is modified to swap pages to another server, thereby improving memory utilization in datacenters [7, 8, 13, 14]. Another approach involves splitting the operating system into monitors, each responsible for managing specific resources [15]. Additionally, in high-performance computing (HPC) systems, a more unconventional method of software disaggregation exists where multiple OS instances manage CPU and memory resources separately within a single server, enabling applications to achieve high performance and scalability [10, 11].

Although software-based disaggregation provides flexibility, scalability, and reliability, these architectures often demand new build environments, OS modifications, application changes to efficiently use resources, thereby impacting developer productivity. For example in order to make use of memory resources efficiently in software based memory disaggregation, AIFM, a library based approach requires application modifications [1]. While specialized OSes [15, 13] show potential to use disaggregated resources efficiently, they do require developers to update their build environments and in some cases also modify their applications [11, 10]. Furthermore, in certain scenarios, the effort required to move to a new OS or modify applications may not justify the porting effort.

In my thesis, I explore automated tools designed to prioritize developer productivity, allowing developers to focus entirely on their applications without the burden of porting efforts required to run on software-based disaggregated architectures. In the following subsections, I discuss two different types of software-based resource disaggregation and the challenges developers face when adapting to these systems. Finally, I briefly explain how my research has contributed to improving developer productivity within these architectures.

1.1 Software disaggregation within a single server in HPC systems

A growing number of applications and runtimes place intense demands on systems which push the traditional hardware and software stack to its limits. The needs of these applications often cannot be met by general-purpose operating systems (GPOSes), either owing to overheads caused by mismatched abstractions [16, 17], system interference and jitter from "OS noise" [18, 19], or unnecessary complexity introduced by a general-purpose OS design [20].

While specialized OSes (SOSes) have been shown to increase performance, in some cases significantly [21, 22, 23, 16] one of the biggest challenges facing their widespread adoption is their non-conformance to POSIX or the Linux ABI. This means that for users to take advantage of a new OS, developers must first port applications to work with the kernel's system interface.

One approach to ameliorate this situation involves *delegation* of a portion of the

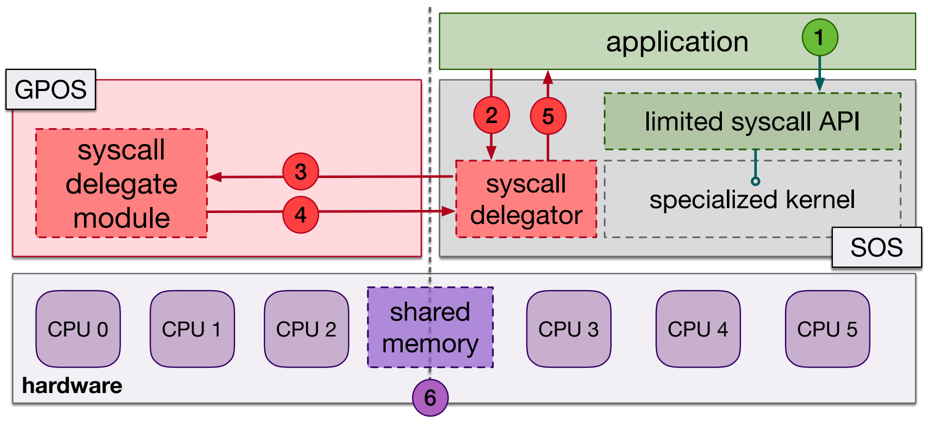


Figure 1.1. High-level architecture of a multi-OS system where the CPU/memory resources of a single server are disaggregated between multiple operating systems.

system call interface to another OS. This approach, sometimes referred to as a multi-kernel or co-kernel [24] setup, partitions the machine (either virtually [17, 25, 26] or physically [24, 27, 28, 10, 11]) such that different OSes control different resources. Usually this means that a GPOS (such as Linux) acts as the *control plane*—setting up the execution environment, launching applications, and handling system services—while an SOS acts as the *data plane* or *compute plane*. The rationale is that the majority of the application's execution will be in the compute plane, and any system services unsupported by the SOS will be delegated to the GPOS via some forwarding mechanism. Multi-kernel systems are an unconventional form of software resource disaggregation within a single server. Figure 1.1 describes the architecture of software disaggregation within a single server often used in HPC systems to achieve high performance [29, 30].

Adapting to a new OS environment may require significant developer effort which may not always justify the gains from it. While others have presented empirical analysis of delegation [31], and several multi-OS designs exist [24, 10, 11, 32], there had been no prior attempt to model these environments analytically. This presents an opportunity, as "bounds and bottleneck" analysis can provide valuable insight and intuition for novel computing paradigms. We draw inspiration from prior work such as Amdahl's law [33] and its successors [34, 35] provided keen insight on the limitations of parallel program performance at a time when parallel systems and algorithms were emerging.

Our work on modelling speedups in multi-kernel systems [36, 37] empowers developers with an automated tool to gain insight whether their application would benefit from such complex systems, before having to modify their software environments. I will describe our speedup models in more detail later in Chapter 3.

1.2 Software resource disaggregation across servers in datacenters

Another form of software disaggregation involves extending DRAM on a remote server connected to the local machine with a high-performance interconnect to fetch additional memory from other servers. One example of such a *far memory* tier is *remote memory*, alternatively referred to as *disaggregated memory* [38]. Remote memory systems accommodate memory-constrained applications by allowing workloads to scale across machines rather than requiring overprovisioning using expensive, large-memory hardware. This reduces ownership costs [39] and mitigates application crashes from unmet memory demands.

Figure 1.2 is an example of a far memory disaggregated system in datacenters, where memory is disaggrgeated across servers within datacenters. Software-based remote memory are generally implemented by two primary techniques: *kernel-based* and *library-based*. The kernel-based approach modifies the OS paging subsystem [40, 8, 7], achieving *programmer transparency*: the application developer gets the advantages of kernel-based approaches for free; even unmodified binaries can benefit from remote memory. Fastswap is a notable example that uses a modified Linux swap subsystem to leverage memory on a remote server using RDMA [7]. The programmer transparency of the kernel-based approach comes at a cost, however. For example, page fault overheads in the kernel impose a performance penalty on applications relative to using only local memory [1]. The hardware

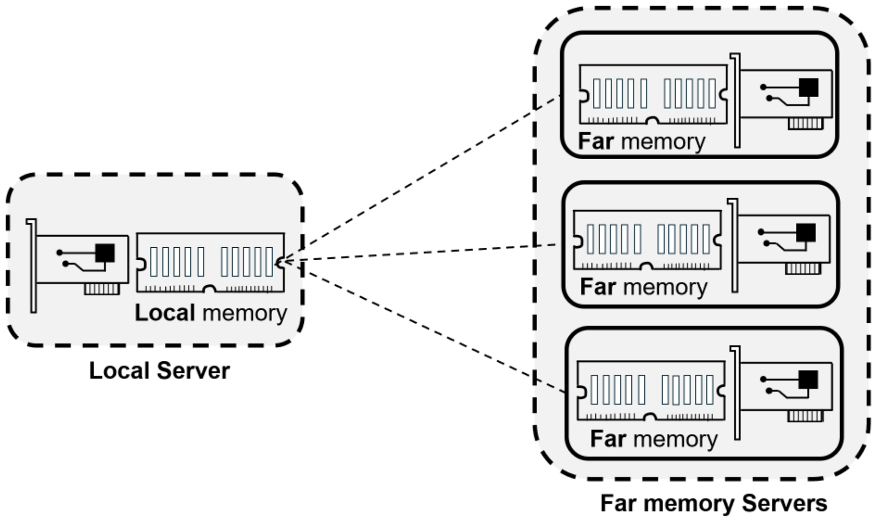


Figure 1.2. High-level architecture of a far memory system where the memory of the local server is extended to use unused memory from other servers within a datacenter.

page fault cost creates a fundamental limitation on performance.

The library-based approach to far memory is an important alternative, where developers use modified (or custom) libraries that include data structures designed to leverage remote memory, at granularities appropriate for the application, and entirely in user space. Application-integrated far memory (AIFM) [1] is the exemplar of this approach. AIFM builds on the Shenango runtime's [41] high-performance user-level tasking and networking to hide remote object fetch latencies using prefetching, concurrent fetch requests, caching, and automatic memory evacuation. AIFM can thus achieve considerably higher performance than Fastswap, especially for fine-grained objects. The performance of the library-based approach trades off for programmer transparency, since the application must be reimplemented to leverage remote memory.

The tension between transparency and performance in the kernel-based and library-based approaches creates an opportunity for a third alternative: *compiler-based*. We argue that modern compiler analysis and transformation techniques make it possible to simultaneously achieve programmer transparency and performance. Our work TrackFM [42] is the first to show how modern compiler analysis can provide performance without requiring any application changes in these architectures. TrackFM enhances developer productivity by automatically transforming code to memory disaggregated architectures, this enables developers to focus solely on their application and they do not have to worry about modifying their application. I will describe TrackFM in more detail in Chapter 4.

We discovered that while TrackFM highlights that compiler analysis can be leveraged to automatically transform applications and use far memory efficiently, The policies on what objects should be remoted, prefetched and evacuated are determined conservatively. In certain cases when an application has enough local memory much of the instrumentation costs of TrackFM can be avoided if policies such as what object are remoted are determined at runtime instead of compile time. For example in the TrackFM compiler, all memory objects are assumed to be remoted. This decision is made conservatively as the TrackFM compiler lacks information at compile time to make the right decisions. Profiling compilers like Mira [43] on the other hand leverage profiling to determine the right policies for remoting, prefetching and evacuation. While profiling does help address this limitation for certain applications, it may require several runs of the application to make policy decisions efficiently, and can also be expensive.

We draw inspiration from prior library based approaches [1] that leverage user hints to identify and annotate remote data structures which enables user runtimes to make policy decisions dynamically. However, without user hints or profiling, the capability to make far memory policy decisions efficiently is challenging in compiler based approaches. First, source code level information (e.g., custom data structures) is lost during compilation from source language. The information of data structures and their access patterns are critical to determine prefetching and eviction policies dynamically in far memory runtimes like AIFM [1]. Second, the compiler analysis is limited to static information available at compile time and does not take into account the dynamic behavior of far-memory applications. For instance, the compiler cannot know at compile time whether an object of a linked list is local or remote, which restricts the compiler's ability to make informed policy decisions.

In CARDS, we overcome the first limitation by leveraging data structure analysis [44, 45] to automatically recover data structures from source code at compile time. We overcome the second limitation by co-designing the compiler and runtime, i.e., we pass the compiler identified data structures to the runtime using data structure handles. This enables the runtime to make policy decisions per data structure dynamically and avoids the need to make conservative policy decisions at compile time. I will describe CARDS in more detail in Chapter 5.

CHAPTER 2
THESIS

Software-based resource disaggregated architectures should prioritize developer productivity alongside efficient resource utilization. Leveraging low-level system software components such as compilers, operating systems, and analytical models can significantly enhance productivity within these complex systems. By building automated tools, developers can focus exclusively on their applications, abstracting away the intricacies of resource management and optimization in software-based disaggregated environments.

CHAPTER 3

ANALYTICAL SPEEDUP MODELS TO PREDICT SPEEDUP IN SOFTWARE DISAGGREGATED SYSTEMS

I describe below our speedup models along with a multi-OS emulator which will provide insights into application performance in a single server OS disaggregated system (multi OS) environment without developers having to port their application.[1]

3.1 Analytical Speedup Model Design

Multi-OS environments are arranged such that compute intensive portions of an application run atop a specialized OS (compute plane), and system services not supported by that SOS are *delegated* to a general-purpose OS (control plane), which handles the calls and returns the results. There are two primary concerns when considering this setup: which calls to forward; and *how* to forward them. The first concern might depend on the nature of the calls. For example, if there is no filesystem support in the SOS, system calls like `read()`, `write()`, and `open()` must be delegated. In HPC environments such I/O offload is often employed to reduce load on the parallel file system (PFS) induced by concurrent client requests from compute nodes. Instead, filesystem requests are delegated to an I/O node. In other cases, the choice of which system calls to delegate might hinge on time and resources available to the OS developer. In the interest of time, he or she might implement commonly invoked system calls in the SOS to optimize for the critical path, but choose to delegate those invoked infrequently. Gioiosa et al. showed how this choice can be guided by empirical analysis [31].

The second concern (regarding the delegation *mechanism*) depends on the capabilities of the hardware and software stack, and on the use case. For example, modern Linux kernels allow for offlining a subset of CPU cores which can then be controlled by an SOS.

[1]The content of this chapter is gathered from published research and is reprinted, with permission, from [46, 37]

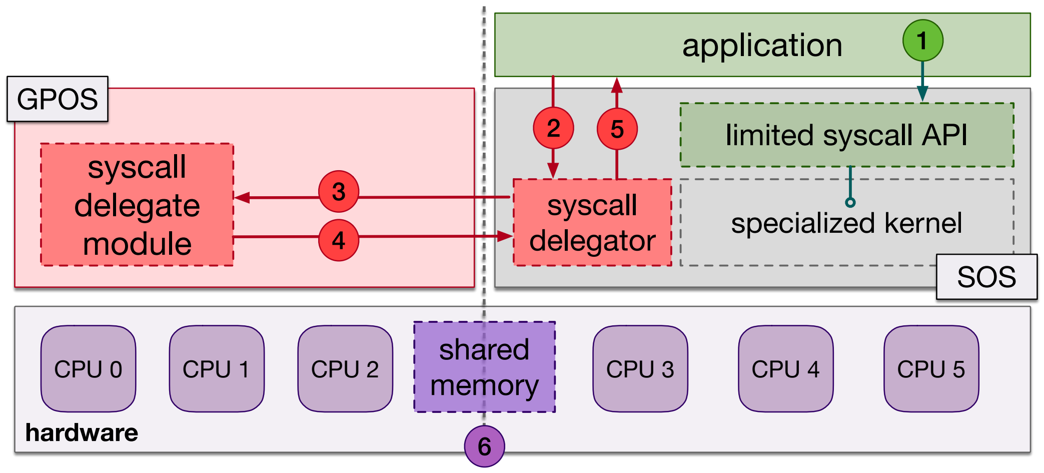


Figure 3.1. High-level architecture of a multi-OS system leveraging the delegate model.

IHK/McKernel [11], Pisces [24], FusedOS [32], and mOS [10] leverage this feature. In this case, because the two OS kernels run on the same node, delegation can occur between an SOS and the GPOS using shared memory. If the SOS and GPOS are running on separate nodes, however, delegation must occur over the network, which involves marshalling arguments and initiating remote procedure calls (RPC) between nodes, unless the system supports distributed shared memory. Remote delegation is necessary for the I/O offloading example mentioned previously. In cases where the machine is partitioned using virtualization, as in Libra [26] and Hybrid Virtual Machines [17], delegation may occur either via VMM-managed shared pages or via explicit hypercalls from guests.

Figure 3.1 shows an environment that supports *local* system call delegation in a hexa-core machine. The machine is physically partitioned between a GPOS and SOS such that they own a subset of the physical resources (memory and processors). In this case, the GPOS runs on cores 0–2 and the SOS runs on cores 3–5 (the compute cores). Memory is assumed to be partitioned such that the physical address space is split between them (6). When an application invokes a system call supported by the SOS (1), the SOS kernel handles it directly. However, when the application invokes an unsupported system call, it vectors to a handler in the specialized kernel (2), which communicates with a component

(3) in the GPOS (such as a kernel module), which fields the original request. In this case, the communication between the SOS and GPOS is facilitated by a shared page of memory. This delegation process involves some subtlety, as the context of the calling (delegator) process must be mirrored by the handling (delegate) component, and the system call handler service might exist in the GPOS kernel or in a user-space process running atop the GPOS, as in User-level Servers in mOS [10]. After the delegate handles the system call, it sends back the results to the SOS (4), which then returns the result to the application (5).

While there are technical differences between existing multi-OS systems, they all share two primary characteristics that we will use in modeling them. The first is that they assume some difference in performance when a program runs in the GPOS and in the SOS. The second is that some delegation or forwarding mechanism exists to allow the two kernels to communicate.

3.2 The Delegate Speedup Model

We now present two versions of a model which represents the speedup of an application in a multi-OS environment. For the following, we assume a single-threaded application (more on this in Section 3.5) whose computation portion runs in a specialized (compute plane) OS, and whose system portion (namely, system calls) runs on a general-purpose (data plane) OS. Thus, *all* system calls are initially assumed to be delegated to the GPOS.

3.2.1 Naïve Model.

We begin by outlining the simplest and most familiar scenario, namely where there is no system call forwarding. In this scenario, the program is executed entirely on a GPOS. Let T_{orig} be the execution time of the program in this environment:

$$T_{\text{orig}} = T_{\text{orig}} \cdot p + (1 - p) \cdot T_{\text{orig}}$$

Here p is the percentage of the workload related to system calls. This, for example, could be calculated by dividing the number of instructions executed in the kernel[2] by the total number of instructions retired during the run of the program. Such measures are commonly available from hardware performance counters, but the source of these terms is beyond our scope.

Now we consider a scenario wherein we execute the application in an SOS, but forward all system calls to a GPOS. Let T_{new} be the new execution time, and let γ be a constant factor that represents the speedup from running the *computational* portion of the workload in the SOS relative to running the same portion in the GPOS. We hereafter refer to γ as the *gain* factor of the SOS.

$$T_{\text{new}} = T_{\text{orig}} \cdot p + \frac{T_{\text{orig}}}{\gamma} \cdot (1 - p)$$

Using the familiar speedup ratio (T_{orig}/T_{new}), we arrive at the overall speedup (represented by S_n, where the n corresponds to “naïve”):

$$S_n = \left(p + \frac{1-p}{\gamma}\right)^{-1}$$

This intuitive model has power in its simplicity. Consider the case where $p \ll 1$. This corresponds to a workload that spends very little of its time performing system calls (control plane), and thus spends most of its time computing. We might say that this application has very high “operational intensity,” spending more time in user space than in kernel space. In this case, the application receives a significant benefit[3] from the SOS, and the

[2]This would include transparent system events such as page fault exceptions and interrupt handling.

[3]This, for example, might be due to guaranteeing cooperative scheduling or might

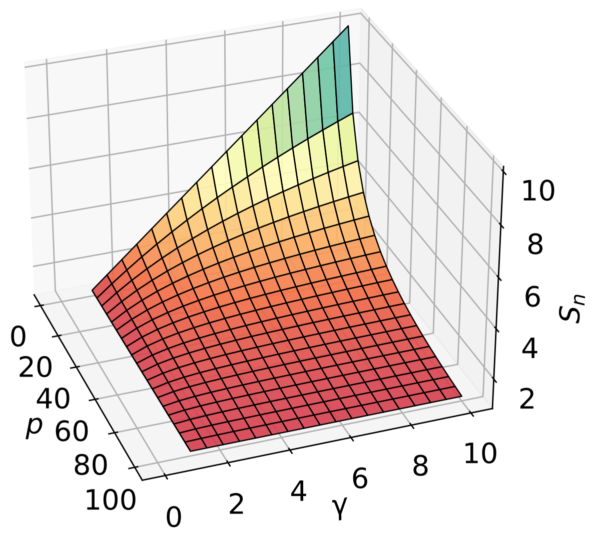


Figure 3.2. Speedup in a multi-OS environment (S_n) given the proportion of time an application spends on system calls (p) and the gain factor (γ) from running in the specialized OS.

overall speedup equation reduces to γ. However, to understand how, e.g., an I/O-intensive application behaves, we observe that as p approaches 1, so does the overall speedup. The intuition here is that a system-only workload will receive *no* benefit from the SOS, and will thus spend most of its time in the control plane. It is important to note just how important p is for this model. Consider that as γ tends towards infinity, this speedup relation tends to $\frac{1}{p}$.

This succinctly captures the bounded speedup of the multi-OS environment, and echoes the insight provided by Amdahl's Law. Essentially what this says is that *even with infinite improvement* of the computational portion of a workload by the specialized OS, the speedup of the application is bounded by how much it relies on the legacy system interface. Figure 3.2 depicts how this model behaves as p and γ vary. Notice how as p grows smaller, we reach perfect linear speedup. The gain factor (γ) is the interesting part of this model, and it very closely resembles the parameter representing parallelism in Amdahl's Law.

be due to an address space setup amenable to (or tailored to) the application.

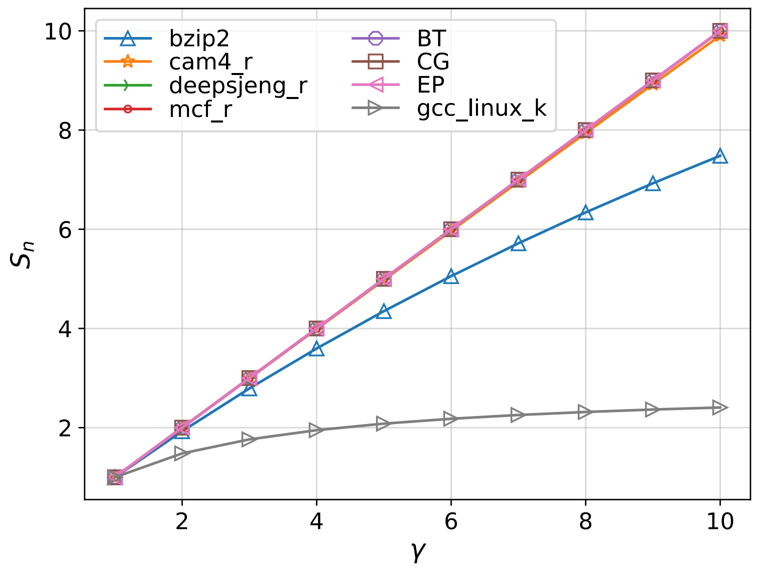


Figure 3.3. Speedup projected by our simple model for SPEC and NAS benchmarks with varying gain factor (γ).

Semantically, however, they are quite different. In practice, we do not expect the gain to be very large (likely < 2), but the interplay between γ and p are still significant for application performance. We show speedups on the order of $10\times$ here to show the general behavior of the model.

One can also use this model from the perspective of an OS kernel developer, in which case it can be used to determine where to focus development efforts. For example, even if monumental effort is spent improving the computational aspect of a workload (e.g., by focusing on developing efficient threading libraries), it might make very little impact if the kernel will run I/O-intensive applications. This of course echoes the oft-stated design principle, “optimize for the common case.”

To understand how this model might translate into real application performance, we first determined p for a selection of benchmarks both from the SPEC CPU 2017 and NAS Serial suite, and projected real application speedup given a fixed gain (γ) factor.

Table 3.1. Empirically determined system call portion (p) for SPEC and NAS benchmarks (class C).

Benchmark	**Description**	p
`gcc_linux_k`	Linux kernel compilation	35.05%
`bzip2`	bzip2	3.74%
`cam4_r`	Atmospheric modeling	0.107%
`deepsjeng_r`	Deep Sjeng chess engine (tree search)	0.0054%
`mcf_r`	Combinatorial optimization	0.00264%
`BT`	Block Tri-diagonal solver	0.000067%
`CG`	Conjugate Gradient	0.00026%
`EP`	Embarrassingly Parallel	0.000152%

To determine p empirically, we used `strace`[4] and the `time` command for a reference run of the individual benchmark. We calculate p by summing the system call times (measured with `strace`) spent in the application and computing its ratio to the total execution time (measured with `time`).

Furthermore, p may vary for a particular benchmark when its inputs are changed. Table 3.1 shows the empirically determined values of p and descriptions for each benchmark we used. Figure 3.3 shows the results of our experiment. Linux kernel compilation (`gcc` in the graph) stresses the system interface the most (due to heavy file I/O), and thus achieves very little speedup, even with a significant initial speedup from the SOS, represented by γ. The other benchmarks are skewed towards computation, and thus achieve sub-linear speedup as γ increases.

3.2.2 Refined Model.

While our naïve model can be used as an intuitive tool, there are several simplifications that make it unrealistic in terms of predicting performance:

[4]`strace -c -f -w -D`

1. The cost of forwarding system calls is ignored. In the existing model, this means that we assume *all* of them are forwarded.

2. Different system calls have different costs (in terms of execution time).

3. A given system call will have different costs for different invocations (in most cases determined by its arguments). Consider, for example, the `read()` system call.

4. System calls which are *ported* to the SOS might have different cost than the original GPOS version.

5. It is inaccurate to say that the speedup factor (γ) applies uniformly to all non-system instructions in the program. For example, the SOS environment might have a simplified paging setup (e.g., identity-mapped, 1GB pages) which significantly reduces TLB misses for instruction fetches and loads and stores, but integer/floating point instructions will be unaffected unless they involve memory references.[5]

6. Setups where there are more than one GPOS and more than one SOS are not considered.

In this Section, we refine our model by addressing (1), (2), and (3) above. We intend to refine the model further in future work to account for the remaining simplifications.

We first must capture the fact that different system calls can have different costs ((2) and (3)). Let $\mathbb{S} = \{s_1, s_2, \ldots, s_n\}$ be the set of all system calls invoked during a particular run (with fixed inputs) of program P. We introduce a function $g : \mathbb{S} \rightarrow \mathbb{R}$, such that $g(s)$ gives the absolute time taken for *all invocations* of system call s in the GPOS for the run of program P. For example, if one program run contained several invocations of `mmap()`

[5]This is unless, of course, the instructions cause an exception or involve addressing modes that necessitate a memory reference.

(which is common), $g(mmap)$ will represent the time taken for *all* such invocations[6] when run on a general-purpose OS. This function also includes time spent in the kernel due to, e.g., blocking system calls.

Let C represent the absolute time taken by the computational portion of the program (that is, all instructions *not* executed in the context of a system call). We can then calculate the total execution time in the default case, where the program runs entirely in the GPOS and no system call delegation occurs (represented by t_{nd}) as follows:

$$t_{\text{nd}} = C + \sum_{s \in \mathbb{S}} g(s)$$

We then must capture the notion of system call delegation. We introduce two functions $f : \mathbb{S} \to \mathbb{R}$ and $b : \mathbb{S} \to \{0, 1\}$. The function $f(s)$ represents the time required to forward system calls, defined as:

$$f(s) = 2 f_c n(s)$$

Here f_c is a constant that represents the *base* forwarding cost for all system calls using a particular forwarding mechanism, and the function $n : \mathbb{S} \to \mathbb{N}$ represents the number of times system call s is invoked. f_c is scaled by a factor of two to account for the round-trip from the SOS to the GPOS. That is, a system call is forwarded from the SOS to GPOS, executed on the GPOS, and the results are sent back to the SOS, so the forwarding overhead is incurred twice. f_c will vary widely depending on the software mechanisms which implement forwarding and the underlying interconnect over which system calls are forwarded.[7] For example, for delegation over shared memory, f_c would likely be in the

[6]That is, the sum of the execution times for all mmap(). invocations.

[7]Here we make the simplifying assumption that this cost is *independent* of the system

ns to μs range For delegation over a network, this number might be closer to several μs or several ms, depending on the network characteristics. Note that delegation over the network may involve more complex and asymmetric forwarding costs. For example, the forward trip may involve marshalling system call arguments, and the return trip from the GPOS might only involve a single integer return value. We do not currently take these complexities into account in our model.

The second function, $b(s)$ is a boolean predicate function[8] which tells us whether or not a particular system call is delegated:

$$b(s) = \begin{cases} 0, & \text{if } s \text{ is not delegated} \\ 1, & \text{otherwise} \end{cases}$$

Recall that the program primarily runs in the context of the SOS, and so receives some performance benefit (represented before by the factor γ) as a result. Thus, as before, we only apply γ to the computational portion of the workload (C). For the system call portion of the workload, we must differentiate between delegated system calls and local system calls (those which have corresponding implementations in the SOS). We can represent the absolute time taken by all *local* system calls (t_{local}) by introducing another function g' which captures this notion. We use $g'(s)$ to represent the time taken for all invocations of system call s given the custom version of s implemented in the SOS.

$$t_{\text{local}} = \sum_{s \in \mathbb{S}} (1 - b(s)) g'(s)$$

call itself, but this is not strictly true. A forwarding mechanism that uses marshalling (e.g., over a network) will incur more forwarding costs for a system call with more arguments.

[8]We could also refer to this as a characteristic function or an indicator function.

We then represent the absolute time taken by all *delegated* system calls (t_{remote}) as follows:

$$t_{\text{remote}} = \sum_{s \in \mathbb{S}} b(s)(g(s) + f(s))$$

We can now calculate the total time taken (t_d) for a setup where *some* system calls are delegated:

$$t_{\text{d}} = \frac{C}{\gamma} + t_{\text{remote}} + t_{\text{local}}$$

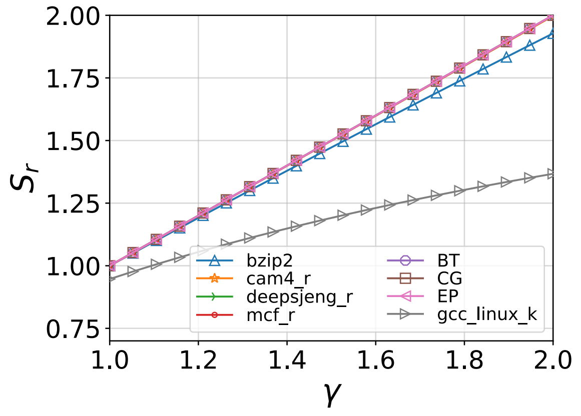
Thus, we can represent our new speedup (S_r) as

$$S_{\text{r}} = \frac{t_{\text{nd}}}{t_{\text{d}}}$$

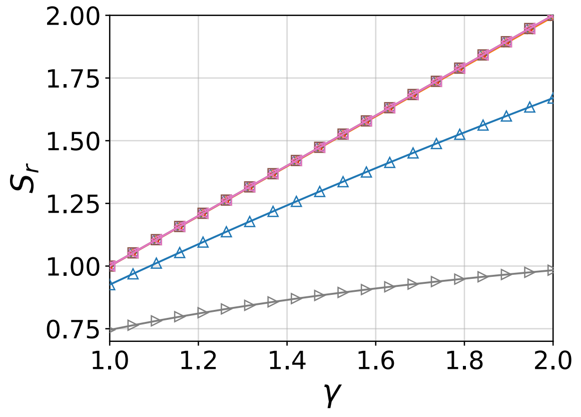
Expanding this out, we get

$$S_{\text{r}} = \frac{C + \sum_{s \in \mathbb{S}} g(s)}{\frac{C}{\gamma} + \sum_{s \in \mathbb{S}} b(s)(g(s) + f(s)) + (1 - b(s))g'(s)}$$

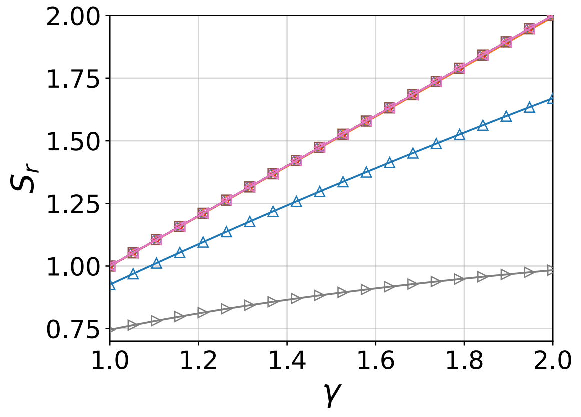
Intuitively, the more system calls that are forwarded (those which have $b(s) = 1$), the more overhead is incurred, and speedup is curtailed. Notice that in the denominator, the time taken by the computational portion is scaled by a factor of $\frac{1}{\gamma}$. An ideal scenario would have $g'(s)$ take *less* time than $g(s)$, meaning that an implementation of a system call in the SOS would be more efficient than its counterpart in the GPOS. However, going forward, we make the simplifying assumption that $g(s) = g'(s)$, so that both implementations take the same amount of time.



(a) min. frequency



(b) max. frequency



(c) random

Figure 3.4. Projected speedup when the gain (γ) varies for SPEC CPU 2017/NAS benchmarks. This assumes a fixed forwarding cost (f_c) of 10 μs and a fixed percentage (90%) of forwarded system calls. Three schemes for choosing which system calls to forward are shown. From left to right, we select system calls by least frequently invoked (a), most frequently invoked (b), and randomly (c).

Figure 3.4 illustrates speedup projections (represented by S_r) using our refined model for a subset of the benchmarks shown in Table 3.1. We initially fix the forwarding cost, f_c, at 10 μs. This is representative of a scenario where forwarding occurs over shared memory. We vary γ to illustrate the effects of running the application in the SOS.

Each benchmark in the suite invokes a different set of system calls, and here we are interested in seeing the effects of choosing different sets of system calls to forward. In this case, we show three scenarios. A fixed proportion of 90% of system calls are forwarded. This proportion reflects what we have seen on mOS, a real multi-kernel system. In each graph we vary *which* system calls constitute that fixed proportion. In the first two scenarios, system calls are chosen according to how many times they are invoked. Figure 3.4(a) shows the projected speedup when system calls invoked *infrequently* by the application are chosen to be forwarded. This is the most ideal scenario, as the forwarding overheads will not be incurred often. Note that in IHK/McKernel, another multi-kernel system we studied, roughly 30% of system calls are forwarded. If that proportion were used here, we would not see much effect on projected speedup since most of these calls are infrequently invoked (see Figure 3.9 and related discussion). Figure 3.4(b) shows the speedup when we choose system calls which are invoked *most frequently*. This is not a forwarding policy one should choose, but is shown here for comparison. Finally, Figure 3.4(c) depicts the results when we make a random choice. Note how different benchmarks are affected differently by the forwarding schemes. Both SPEC CPU and NAS benchmarks achieve a high speedup no matter the scheme. This is because of the generally low system call activity in these benchmarks (as shown in Table 3.1). However, `bzip2` shows significant difference when we compare the "min. frequency" case with the "max. frequency" case. To see why, it is illustrative to study Figure 3.9, which shows a breakdown of system call usage for several of the benchmarks. In this experiment we traced all system calls invoked by each benchmark, and counted the number of invocations for each individual system call. We report these counts using a CDF. A point on this figure thus represents the percentage of system calls that have been invoked fewer times than the value on the horizontal axis. Looking at the CDF for `bzip2`, it is clear that its speedup is curtailed because it uses a small set of system calls often (this particular application invokes `read()` and `write()` almost exclusively). It is thus critical that those system calls are *not* forwarded. When they are, as in Figure 3.4(b), performance is severely affected. It is also clear from the figures that applications which have a more varied system call distribution will be less affected by selective forwarding schemes. More generally, workloads showing system call profiles with more statistical structure will be more amenable to selective forwarding schemes. This aligns with intuition and prior experimental results from Gioiosa et al. [31].

In Figure 3.5, we choose three benchmarks `bzip2` (skewed system call distribution), Linux kernel compilation with `gcc` (uniform distribution of system calls) and one from the NAS suite, `BT` class C (compute intensive with very low system call activity), and show with surface plots how their projected speedups change as we vary both the forwarding cost (f_c) and the gain factor from execution in the SOS. Here we forward 90% of system calls (those invoked most *infrequently*). Note again the log scale on the f_c axis, so the lower end of the scale indicates forwarding costs in the nanosecond range, the middle approaches milliseconds, and the higher end approaches roughly ten seconds. Along the γ axis, all benchmarks achieve a speedup, but note that from left to right these benchmarks have characteristics less amenable to system call delegation, respectively. The `BT` benchmark achieves the highest speedup, both because it has a small system call portion, and because that portion involves very few system calls that are invoked in general. Linux kernel compilation with `gcc` has the highest system call portion and a more uniform distribution of system calls, which leads to a curtailed speedup, resulting in a flat surface.

For `gcc` as the forwarding cost increases, we observe negative speedup (i.e., GPOS performs better than SOS), when speedup values go below 1 we trim the values in Figure 3.5 for better visualization.

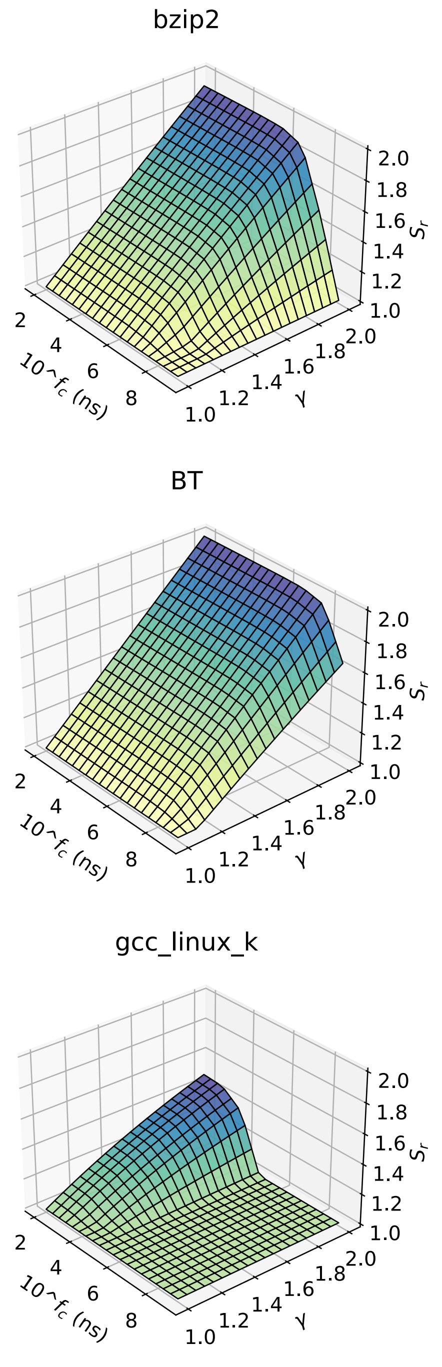


Figure 3.5. Projected speedup as the gain (γ) and forwarding cost (f_c) are varied for SPEC benchmarks. The proportion of forwarded system calls is fixed at 90%, and which calls to forward is determined according to those least frequently invoked (min. frequency). γ is fixed at 2.

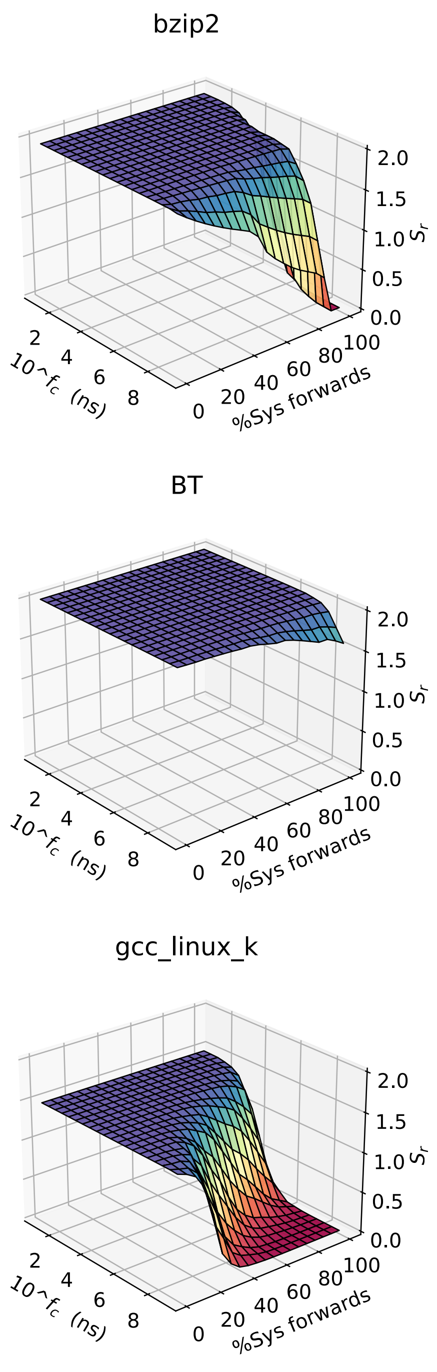


Figure 3.6. Projected speedup as the forwarding cost (f_c) and the proportion of forwarded calls vary, with gain (γ) fixed at 2.

Forwarding cost only becomes a significant factor in the `bzip2` and `BT` cases when it becomes greater than tens of milliseconds. This is more tolerance to forwarding overheads than we expected, and indicates that in many practical cases our naïve model may be sufficient.

Figure 3.6 shows another perspective on forwarding overheads. Here, we show the speedup projected by our model as we vary both the forwarding cost (f_c) and the percentage of forwarded system calls (assuming that percentage consists of infrequently invoked system calls). We make two observations, both of which again align with intuition. The first is that workloads with more skewed system call distributions are more amenable to delegation, even when a relatively large portion of the system call interface is forwarded. The second observation is that when these workloads are *not* properly accounted for (i.e. when the *wrong* calls are forwarded), the performance degradation is dramatic, as shown in the curve for `bzip2`. An interesting note about this visualization is that the "topography" of the speedup curves directly reflect the structure in the application's system call invocation trace.

`bzip2`'s surfaces has a steeper drop-offs when an increasing number of system calls are forwarded, indicating a heavy skew in the system call distribution (cf. Figure 3.9). The smoother "rolling hill" of the BT benchmark indicates a small amount of syscall activity, and the more pronounced drop-off of `gcc` indicates higher, but more uniform syscall activity. Figure 3.6 also shows us that compute-intensive benchmarks (like `BT`) are most amenable to multi-kernel environments, since they are mostly unaffected by forwarding overheads.

In Figure 3.7, we vary the gain and the proportion of forwarded calls. The interesting point here is that with the fixed forwarding cost of 10 μs, we only see an effect for `bzip2` when almost *all* calls are forwarded (thus capturing the frequently invoked `read()` and `write()` calls). Kernel compilation and `BT` are largely unaffected by such

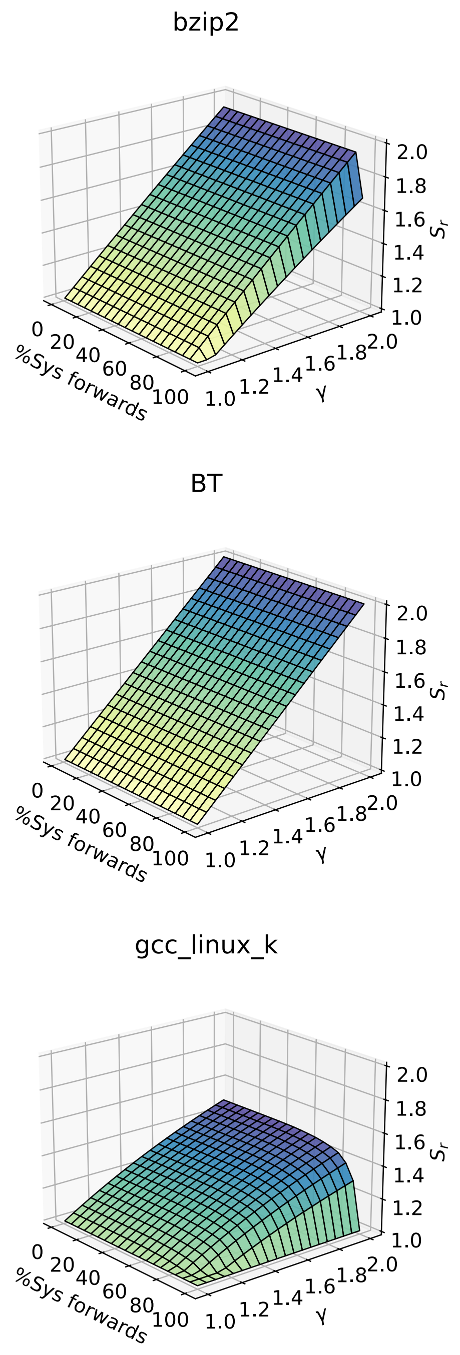
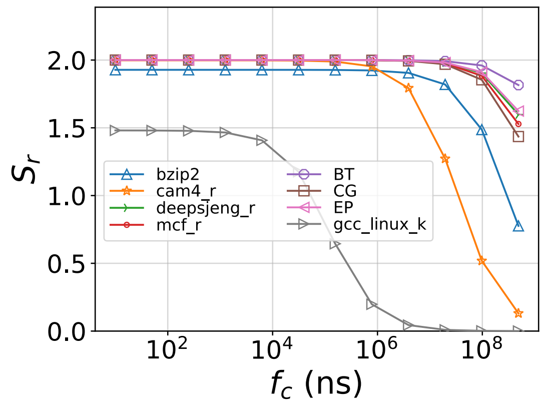


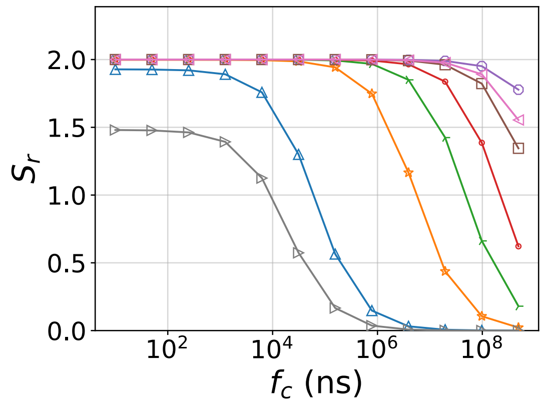
Figure 3.7. Projected speedup as the gain (γ) and the proportion of forwarded calls are varied. The forwarding overhead (f_c) is fixed to 10 μs.

small forwarding overheads.

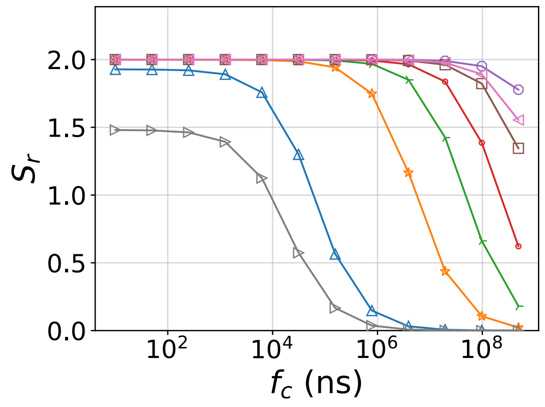
In Figure 3.8, we again show the effects of different forwarding policies, but as a function of varying forwarding overheads. `mcf_r`, `deepsjeng_r` and `bzip2` have steep



(a) min. frequency



(b) max. frequency



(c) random

Figure 3.8. Projected speedup as the forwarding cost (f_c) varies. We fix the gain factor (γ) to 2 and assume that 90% of system calls are forwarded.

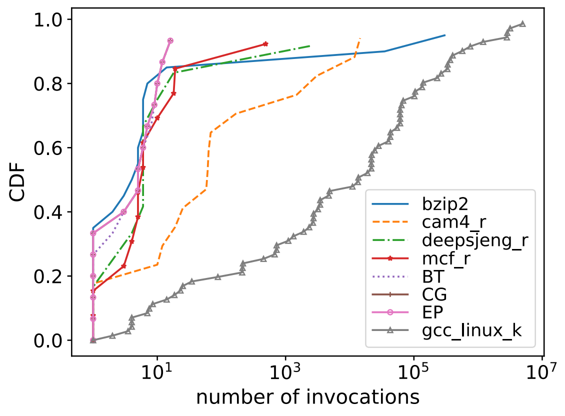


Figure 3.9. System call profile for selected benchmarks. Note the log scale on horizontal axis.

drop-offs in speedup when the wrong set of system calls is forwarded. When *infrequently* invoked system calls are delegated to the GPOS, forwarding overheads must reach several hundreds of milliseconds before making a significant impact.

3.3 Approximate Speedup Projections

The primary obstacle in measuring application performance directly in a multi-OS environment is the OS development burden required to implement functionality in the SOS. The simplest case is when the *entire* system interface is delegated to a GPOS. In this case, the application benefits solely from the properties of the execution environment provided by the SOS (e.g., simplified, deterministic paging and fine-grained interrupt control), and no development effort is spent porting system calls to the SOS. However, this scenario is not ideal, as the results from the previous Section indicate. However, it would be useful to project performance *before* undertaking the development effort to run an application directly on a multi-OS system. In this Section, we describe our multi-OS emulator, called `mktrace`, that enables this projection, allowing users interested in multi-OS setups to perform a kind of "what-if" analysis. Users can run their unmodified programs using our tool to project performance in a multi-OS setup without investing in a porting effort. We believe our tool can provide key insights to developers, in particular those who have no prior experience with multi-OS systems.

Note that this tool does not actually leverage two separate operating systems. Instead, we leverage a Linux kernel module that employs a kernel thread on the *same OS* to serve as a delegate (standing in for the control-plane OS); this delegate fields system call requests from a running process. However, the delegation mechanism in `mktrace` is similar to real multi-OS systems. For example, in the case of IHK/McKernel [11], for every process running on light weight kernel (LWK), a proxy process is created on the Linux side to handle delegated system calls. At a high level, the proxy process acts in a similar capacity to our service threads in `mktrace`. One difference is that a proxy process in a real multi-kernel setup runs in user-space, which requires an additional context switch to handle delegated system calls. Using our tool, a configurable subset of system calls are intercepted by the kernel, which forwards them to this delegate thread with a tunable forwarding cost. With this architecture we can experiment with different delegation schemes to determine performance *without* spending effort porting an application to a new OS. Our emulator tool, called `mktrace`, is freely available online,[9] runs on Linux, and only requires that users load a kernel module before using it.

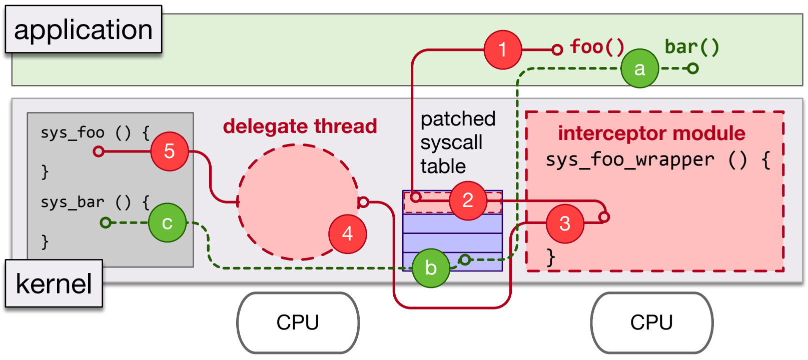


Figure 3.10. High-level overview of `mktrace`.

The primary technique used by our tool is *system call interposition*. This technique has been used primarily for secure monitoring of kernel activity, both using in-kernel or

[9]`https://github.com/hexsa-lab/mktrace`

user-level approaches [47, 48, 49, 50] and out-of-kernel by leveraging an underlying virtual machine monitor (VMM) [51, 52, 53]. Typical interposition tools provide hooking points for system call entry and exit, but we only need to capture entries in order to forward them. Figure 3.10 illustrates how our tool works. After a user loads our system call interceptor (a kernel module), system calls can be selectively forwarded, with tunable forwarding cost. This is achieved by patching the kernel's system call table. In the figure, a regular system call `bar()` ((a)) is invoked, which vectors via a system call table entry ((b)) to the kernel's handler for that system call ((c)). When a forwarded system call `foo()` ((1)) is invoked, our patched system call table entry ((2)) vectors instead to our module ((3)), which mirrors register state (arguments) and the execution environment in the calling process (e.g., address space). Our module then forwards the system call to a *delegate thread*, backed by a separate kernel thread on a separate CPU ((4)), which spins for a configurable amount of time (representing the forwarding delay), and then invokes the kernel's original system call handler ((5)). The results of the system call execution on the delegate kernel thread are sent back to the calling process and execution continues normally.

3.3.1 Experimental Setup.

We conducted our experiments on a system called *tinker*, which has a 2.2GHz Xeon E5-2630 CPU with 10 cores and 48 GB RAM. It runs Centos 7.7 with stock Linux kernel version 3.10.0-1062. Hyperthreading is disabled, and the BIOS is configured for a static power profile (maximum performance) to mitigate measurement noise from DVFS.

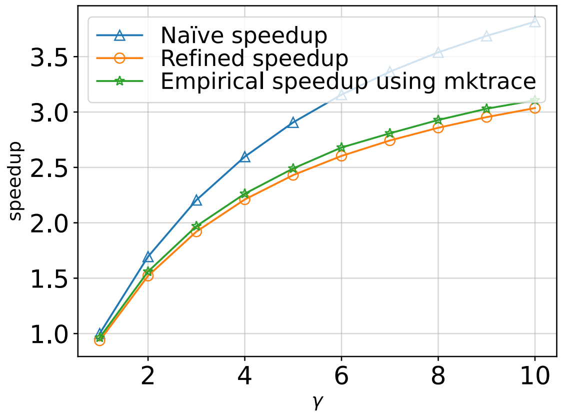
For these experiments, we selected several benchmarks from the initial set: `bzip2`, Linux kernel compilation (kernel version 5.1.4) with `gcc`, the SPEC CPU suite, and the NAS benchmarks (currently the serial versions).

In this Section we are not actually running applications in a real multi-OS system (cf. Section 3.4), so we cannot empirically observe the speedup factor γ given by running the application in the SOS. To approximate this factor, we designed a synthetic benchmark which performs phases of variable amounts of computation followed by phases of fixed system call invocations. The computation is a simple Monte Carlo calculation of π, dominated by floating point operations. To artificially induce a speedup, we simply vary the amount of computation (by reducing the number of trials in the approximation) according to a γ value. Thus, a higher value of γ is approximated by a concomitant reduction in the amount of work done in the computation phase. The benchmark uses a synthetic system call profile derived from traces gathered from `bzip2` (skewed system call profile, I/O intensive) and `BT` (uniform distribution of system calls, compute intensive) using `strace`.

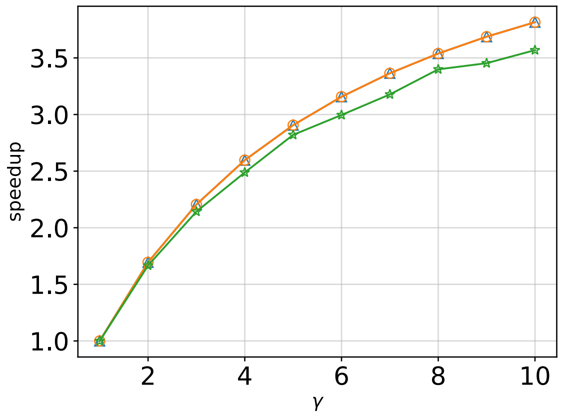
3.3.2 Experiments.

We first run the benchmarks described above in a standard Linux environment and then using our `mktrace` tool, which approximates delegation. We measured the minimum overhead of forwarding using this mechanism (represented by f_c in previous Sections) to be 6.3 μs, measured with 1000 trials. This is quite close to the forwarding costs incurred by real multi-kernels, as we will see in Section 3.4. In all cases, because of the small forwarding overhead, there is very little impact on performance. The largest overhead we observed for `mktrace` was less than 1% across all benchmarks when using the "min. frequency" syscall profile.

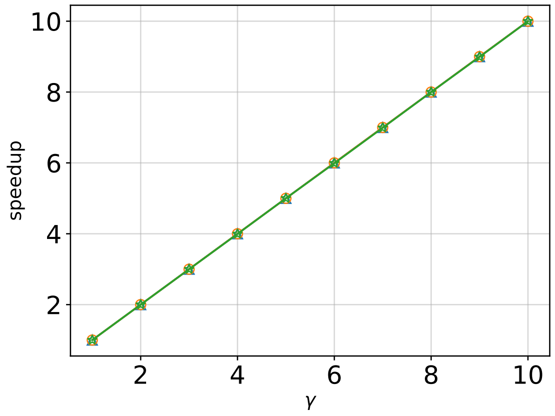
As described above, we now attempt to incorporate the gain factor (γ) by approximating gain using a variable amount of computation. Figure 3.11 shows the projected speedup from both the naïve model and the refined model for the synthetic benchmark compared to the speedup empirically determined with `mktrace`. In this figure, we see how the measured speedup of the benchmark changes with a varying gain factor. The curve labeled "empirical speedup using mktrace" represents speedup relative to the default setup on Linux without any system call delegation.



(a) bzip2 profile; 90% forwarded



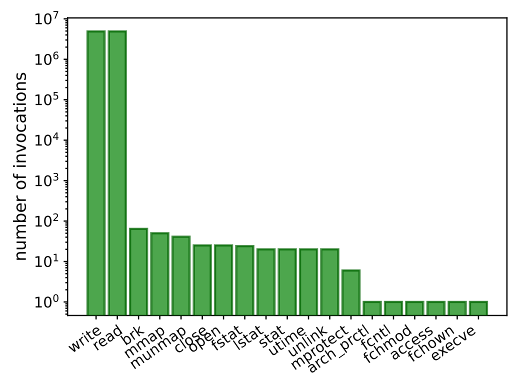
(b) bzip2 profile; 30% forwarded



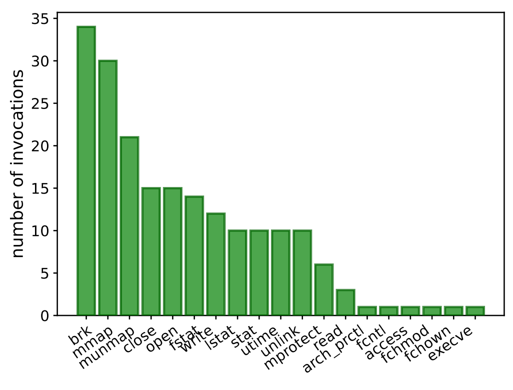
(c) BT profile; 90% forwarded

Figure 3.11. Projected speedup when the gain (γ) varies for the synthetic benchmark.

.



(a) bzip2



(b) BT

Figure 3.12. System call profiles for bzip2 and BT, derived with `strace`. Note the log scale on vertical axis of the first figure.

In Figure 3.11(a) we delegate >75% (similar to mOS) system calls with a system call profile derived from a `bzip2` trace. Here the refined model matches the measured speedup closely. In Figure 3.11(b) we only delegate 30% of system calls (similar to IHK/McKernel). We see that for higher γ values the gap between refined speedup and measured speedup increases.

In Figure 3.11(c), the system call profile is derived from BT, a compute-intensive benchmark. The predicted speedup from both models converge in this case because forwarding costs are negligible.

The small gap we see between the predicted speedup and the measured speedup in Figure 3.11(b) we discovered was due to the fixed forwarding cost assumption our model makes. Depending on how the delegation mechanism is implemented, there is actually some non-determinism in this cost (for example due to queueing and thread wake-up latencies).

In order to understand this further, we run the synthetic benchmark (with the bzip2 system call profile) with a forwarding cost derived using three summary statistics: mean, min, and max. The forwarding cost is estimated by taking the time difference between

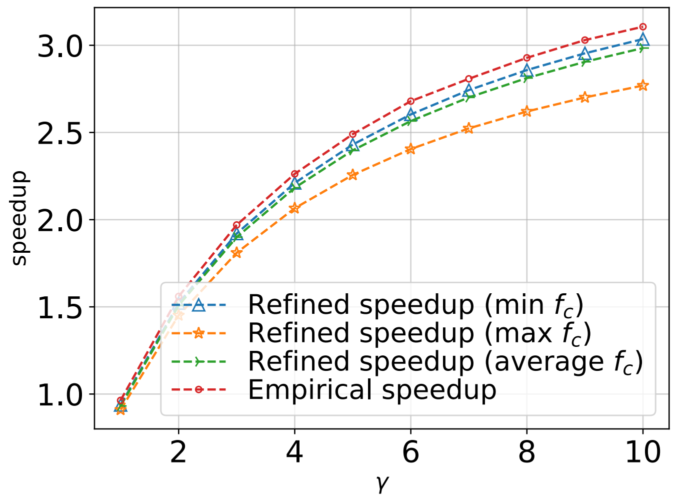


Figure 3.13. Model sensitivity to different summary statistics for derived forwarding cost (f_c).

a system call running with and without `mktrace`. Figure 3.13 shows us that using the minimum measured forwarding cost gives us the best speedup prediction, suggesting a skewed distribution. This is not too surprising, since very rarely kernel events like interrupts and context switches will inflate the forwarding cost.

3.4 Experimental Validation

In this section, we validate our model by comparing its speedup projections with actual execution time on real multi-OS systems. We study two multi-kernel systems that have been deployed in production and that have quite different designs: IHK/McKernel[10] from RIKEN and Intel's mOS.[11] We configure both mOS[12] and IHK/McKernel[13] such that one core (the LWK core) and 15GB RAM are dedicated to the SOS. We select 15 GB as this was the maximum memory we could allocate for a single core in IHK/McKernel on

[10]commit hash 62153.

[11]mOS v0.7.

[12]`lwkctl -c lwkcpus=0.1 lwkmem=15g`

[13]`mcreboot.sh -t -c 1 -m 15`

our *tinker* testbed.

3.4.1 IHK/McKernel.

IHK/McKernel runs HPC applications on a light-weight kernel (LWK) to achieve scalable execution [11], but notably the complete Linux API is available via system call delegation. McKernel acts as the LWK and is primarily designed for HPC; it is launched from IHK, the shim on the Linux side. McKernel retains a binary compatible ABI with Linux and it implements only a small set of performance-sensitive system calls, delegating the remainder to Linux. For every process running on McKernel there is a process spawned on Linux called the *proxy process*. The proxy process facilitates system call delegation. It provides an execution context on behalf of the application so that delegated calls can be directly invoked in Linux. The list of system calls handled by IHK/McKernel, can be found in a kernel header.[14] We use this information in our refined model to estimate the forwarding cost for delegated system calls.

3.4.2 mOS.

mOS uses a different design [10]. While the fundamental concepts of mOS remain similar, mOS incorporates the LWK code directly in a Linux kernel image. The mOS system call delegation mechanism is quite different from the proxy process approach. mOS retains Linux kernel compatibility at the level of its internal kernel data structures, which enables it to migrate threads directly into Linux. mOS implements system call delegation by migrating the issuer thread into Linux, executing the system call, and migrating the thread back to the LWK component. The list of system calls handled by mOS, can be found in a kernel header.[15]

[14]See syscall_list.h, available at `https://github.com/RIKEN-SysSoft/mckernel`

[15]See include/linux/syscalls.h, available at `https://github.com/intel/mOS`

3.4.3 Experiments.

To validate our models experimentally, we selected benchmarks from the SPEC CPU 2017 [54], NAS Class C [55], and LAMMPS [56] benchmark suites, in addition to the CCS QCD miniapp (Class 1) from RIKEN's FiBER Miniapp Suite [57]. We only use benchmarks that were able to run on both multi-OS systems. These are listed in Table 3.2.

We compare the performance of the benchmarks running purely on Linux with multi-OS performance, for both mOS and IHK/McKernel. Both mOS and IHK/McKernel are designed to improve the performance of large-scale parallel applications, not sequential benchmarks. Since our model currently only captures sequential setups, we are not looking for a performance improvement on the benchmarks relative to Linux. Rather, we are looking to validate the projections of our model using real systems. We direct readers interested in multi-OS performance benefits and scaling studies to work by Gerofi et al. [29].

We measure the absolute execution time of the benchmarks on the various platforms for ten trials and report the medians in Figure 3.14. We can see that for most applications, IHK/McKernel and mOS have similar performance to Linux, which again is not surprising given that these benchmarks run sequentially on a single core. We observe that I/O-intensive applications such as `bzip2`, which involves frequent system calls, has an advantage on Linux since the multi-OS systems suffer from forwarding costs on system call-intensive workloads.

Our goal here is to compare these execution times to our models' predictions. In general, one seeking to use our models could derive estimates for model parameters (γ, f_c, and the syscall profile) with microbenchmarking, profiling, or by using reported performance numbers from relevant multi-OS papers or developers. We provide example parameters here. To do so, we use both microbenchmarking and profiling with the benchmark suites to arrive at suitable parameters, which could be used by others evaluating speedup

Table 3.2. Selected benchmarks from the SPEC CPU 2017, NAS Class C, LAMMPS, and CCS QCD benchmarks for model validation.

Benchmark	**Description**
`bzip2`	File compression
`cam4_r`	Atmospheric modeling
`deepsjeng_r`	Deep Sjeng chess engine (tree search)
`lbm_r`	Fluid dynamics
`mcf_r`	Combinatorial optimization
`namd_r`	Molecular dynamics
`parest_r`	Optical tomography with finite elements
`BT`	Non-linear PDE solver (using block tri-diagonal)
`CG`	Estimates minimal Eigenvalue of a sparse matrix
`EP`	Generates independent Gaussian deviates
`FT`	Solves a 3D PDE using fast Fourier transform
`IS`	Sorts small integers using bucket sort
`LU`	Non-linear PDE solver (using Gauss-Seidel)
`MG`	Multi-grid on a sequence of meshes
`SP`	Non-linear PDE solver (using scalar penta-diagonal)
`LJ`	Lennard-Jones atomic fluid simulation
`Chain`	Simulation of bead-spring polymer chain melt
`EAM`	Simulation of Cu metallic solid using EAM method
`Chute`	Chute flow simulation for packed granular particles
`Rhodo`	Simulation of Rhodopsin protein
`CCS QCD`	Lattice quantum chromodynamics miniapp

Table 3.3. Measured round-trip forwarding costs for delegated system calls in IHK/McKernel and mOS.

Benchmark	**Measured $2f_c$ (min., in μs)**	
	IHK/McKernel	mOS
`write_to_file`	6.28	7.56
`fstat`	7.05	7.06
`uname`	8.05	7.60
`write_to_console`	10.36	6.46

for similar workloads.

We calculate the forwarding cost parameter (f_c) for IHK/McKernel and mOS separately by running a delegated system call natively on Linux and subtracting its execution time from the time it takes to run the same system call in the relevant multi-OS system. We take the minimum measured value over 1000 trials. Table 3.3 shows the results (measured in μs) of four delegated system calls. Based on these results, we set the round-trip forwarding cost ($2f_c$) to the smallest measurement observed for each system (6.28μs on McKernel and 6.46μs on mOS). Note that our measurement tool invokes these system calls *directly* using assembly wrappers and the `syscall` instruction to avoid including the costs of user-space system call code (i.e., the syscall wrappers in `libc`).

For the gain parameter (γ), we first determine a *benchmark-specific* gain factor for each of the benchmarks in our suites, then aggregate these using a summary statistic which we will use for the actual γ parameter in the model. To determine a *benchmark-specific* γ value, we plugged in the forwarding cost and the overall speedup measured from benchmark runs into our refined model and solved for γ for each benchmark. Table 3.4 shows our derived, benchmark-specific γ values. For IHK/McKernel most benchmarks have γ in the range of ~0.96–1.1. For mOS, the gain factor ranges between ~0.98 and 1.1. If we exclude `bzip2` (not representative for HPC workloads), γ ranges from ~0.99–1.1

Table 3.4. Derived γ values and percentage of system calls forwarded for both multi-kernels.

Benchmark	**Derived γ**		**Syscalls Forwarded**	
	IHK/McKernel	mOS	IHK/McKernel	mOS
bzip2	0.96	0.98	25%	90%
cam4_r	0.99	1.01	35%	88%
deepsjeng_r	1.00	1.00	25%	83%
lbm_r	1.00	1.01	31%	85%
mcf_r	1.00	1.01	23%	77%
namd_r	1.01	1.00	25%	83%
parest_r	0.98	1.00	36%	79%
BT	1.01	1.01	33%	87%
CG	1.00	1.02	33%	87%
EP	1.00	1.00	33%	87%
FT	0.99	0.99	33%	87%
IS	1.00	1.00	25%	83%
LU	1.00	1.00	33%	87%
MG	1.01	1.01	33%	87%
SP	1.00	1.00	33%	87%
LJ	1.00	1.00	27%	80%
Chain	1.00	1.00	27%	80%
EAM	1.00	1.00	31%	81%
Chute	1.00	1.01	27%	80%
Rhodo	0.99	1.01	27%	80%
CCS QCD	1.01	1.01	21%	91%

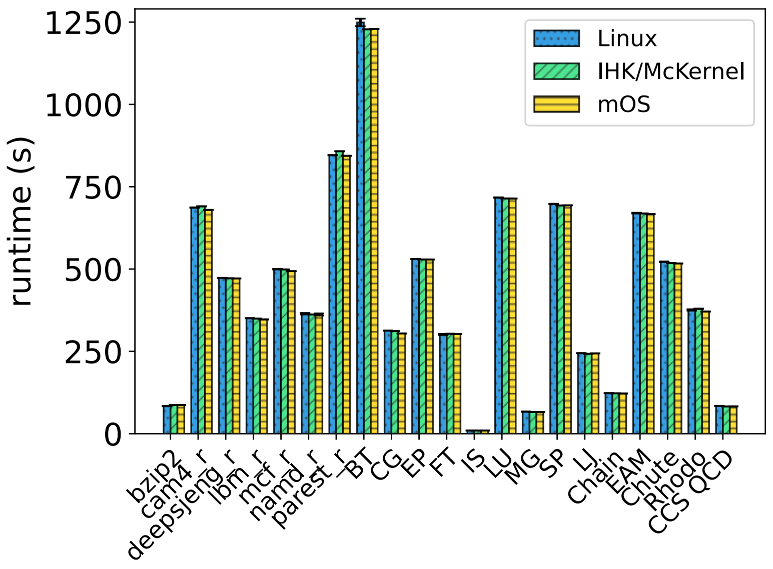
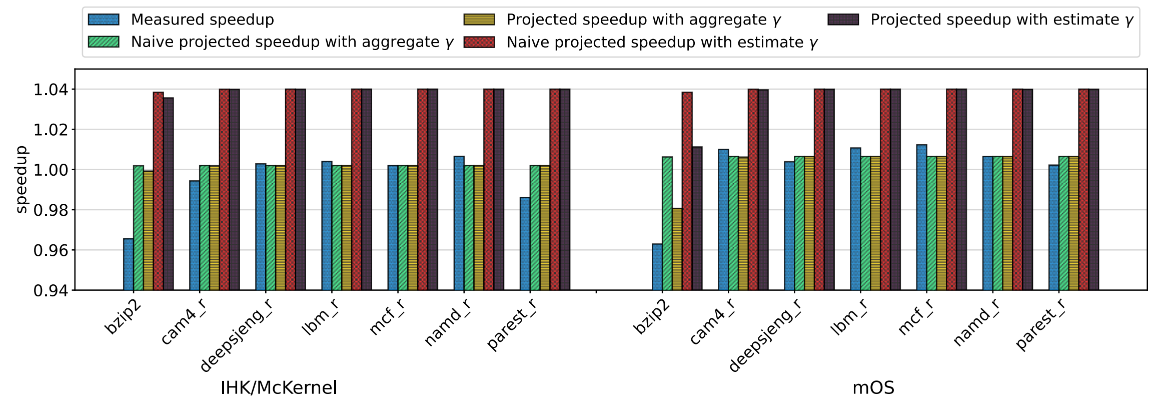


Figure 3.14. Runtime of selected SPEC, NAS Class C, LAMMPS, and CCS QCD benchmarks on Linux, IHK/McKernel, and mOS (lower is better).

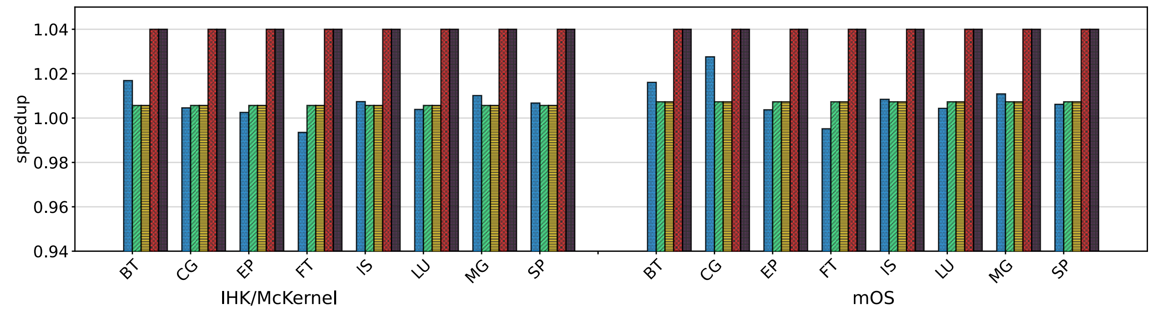
for both multi-OS systems. This is in line with what we expect from single-core, compute-intensive benchmarks.

Figure 3.15 compares our model predictions with measured speedups. Figure 3.15(a) shows the SPEC benchmarks, Figure 3.15(b) shows the NAS benchmark suite, and Figure 3.15(c) shows the LAMMPS and CCS QCD benchmarks. The left halves of the figures show projections and measurements for IHK/McKernel, and the right side depicts mOS. We show speedup projections from our naïve model (every syscall delegated) and our refined model. For both, we use two methods to supply a γ parameter to our model. In the first, we use an estimated gain factor by using empirical measurements from others. In particular, we referred to speedup measurements from Gerofi et al. [29], where the improvements from running in a multi-OS system vary from 4% to 280%.

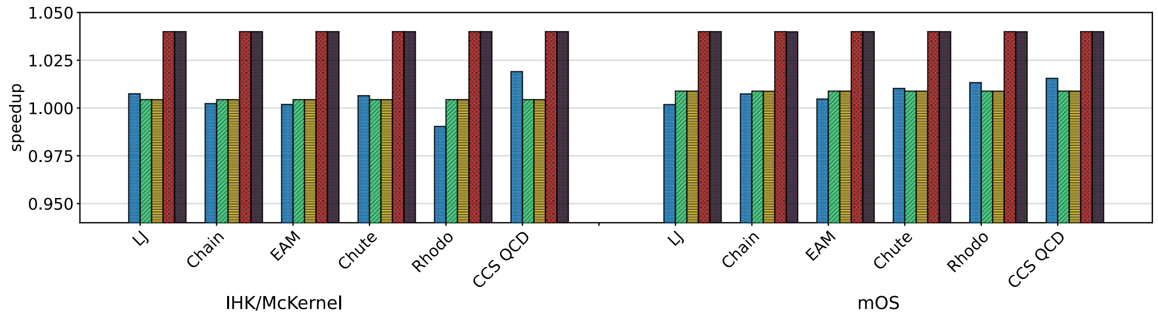
We pick the lowest of these (4% improvement observed in LAMMPS, HACC, and QBOX benchmarks on a single node) as a conservative estimate for γ, as the larger improvements arise from these systems running at scale. We also show projections using a



(a) SPEC CPU benchmarks and bzip2.



(b) NAS benchmarks (class C).



(c) LAMMPS and CCS QCD benchmarks.

Figure 3.15. Empirical speedup on IHK/McKernel (left) and mOS (right) compared to model predictions. We use an aggregate γ value of 1.003 for IHK/McKernel and 1.007 for mOS. We use an estimated γ of 1.04 for both multi-OS systems.

γ value computed as an aggregate (median) of the benchmark-specific values reported in Table 3.4.

In Table 3.4 we also report the percentage of system calls delegated in IHK/McKernel and mOS; we observe that IHK/McKernel delegates a small set of system calls while mOS delegates >70% system calls. In Figures 3.15(a) and 3.15(b) the projected speedup values using an aggregate γ parameter are closest to the mark. This is unsurprising, since this value was derived from the same benchmarks whose speedup is being projected here. The naïve model and the refined model project similar speedups for most benchmarks, since on the whole they invoke system calls infrequently. To determine the accuracy of our models, we compute the mean absolute percentage error (MAPE) of its predictions relative to the measured overall speedups on mOS and IHK/McKernel. The refined model achieves 99.30% accuracy and 99.43% accuracy for IHK/McKernel and mOS, respectively (using the median aggregate γ value). Using parameter estimates from prior work, the refined model achieves 96.18% and 96.79% accuracy, respectively. In practice, a user of this model may not be able to directly determine γ, so the estimated parameters represent a more realistic scenario. One thing to note here is that, provided a fairly close estimate of γ is given, our naïve model will likely suffice to quickly project speedup for compute- and memory-intensive workloads. It can thus be helpful in guiding developers in deciding whether or not to use a multi-OS system.

3.5 Discussion and Limitations

What is clear from the previous Section is that using the models we presented, one can develop intuition for how an application will behave in a software disaggregated system. Notably in the case of compute intensive benchmarks, with forwarding costs up to the microsecond range, the naïve model matches the refined model quite well, indicating that it would be sufficient for single-node setups, as well as multi-node setups with low-latency interconnects. Thus for the single node case, our intuitive model is both simple and

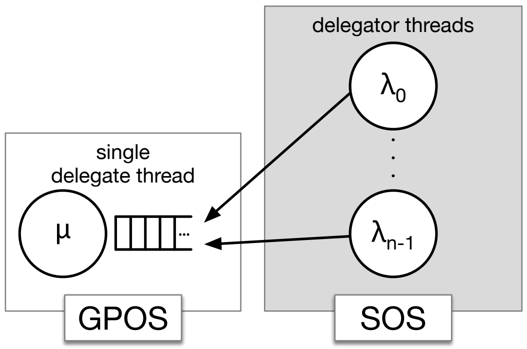


Figure 3.16. Simple $M/M/1$ queueing model for n delegator threads and a single delegate thread.

accurate. While we determine the parameters for our model using manual experiments, a good estimate for them would suffice for coarse speedup projections, a likely use case for our models. Parameters could also be determined automatically by profiling application code or by running a suite of microbenchmarks.

There remain several limitations with these models, however. The primary limitation is that they assume a single delegator thread and a single delegate thread (one on the GPOS and one on the SOS). While this is a reasonable setup for serial workloads, it is unrealistic for parallel applications, where system call requests from the delegator might come from several cores (application threads) or several machines. In this case, a single delegate (GPOS) thread servicing these requests would be inadequate unless the workloads collectively involve few system calls, thus minimizing queueing delays. As we have seen, this is actually not an unreasonable expectation for HPC applications, so an enhancement involving simple queueing models is possible. For example, Figure 3.16 shows a single GPOS delegate thread modeled as an $M/M/1$ queue (Poisson arrivals and exponential service times). Here we assume each of n application threads makes system call delegation requests following a distinct Poisson arrival process, λ_k. By the merging property, these arrivals sum as follows.

$$\lambda_{total} = \sum_{k=0}^{n-1} \lambda_k$$

The delegate thread handles system call requests following exponential service times, with average service rate μ. The λ_k and μ parameters can be easily determined with application traces. Using Little's Law we can then determine the average queueing delay W for system call delegation requests (subtracting out service times):

$$W = \frac{1}{\mu - \lambda_{total}} - \frac{1}{\mu}$$

We can then estimate speedup by incorporating this into our refined model in Equation 3.2.2, adding $W \times n(s)$ to the system call service times $g(s)$, where $n(s)$ is the number of invocations of syscall s and W is the average queueing delay determined above. While a careful analysis of such queueing models is outside the scope of this paper, it is worth discussing their limitations. In workload scenarios that require more than one delegate thread, an initial treatment might extend the model to an $M/M/c$ (multiple server) model. However, the primary issue here is that treating the service rate as a memoryless process ignores confounding performance factors caused by concurrent request streams. Concurrent system call requests coming from two delegator threads may actually both touch the same shared state in the GPOS. For example, in Linux, two concurrent `mmap()` calls will both mutate the region tree in the parent process's task struct. The locking overheads caused by such shared accesses will not be included in a simple queueing model, so it would likely overestimate speedup. Similarly, such a model would not account for low-level hardware overheads sensitive to thread placement, e.g., those due cache coherence traffic and NUMA. The fixed forwarding cost assumptions we make are also a limitation. This assumption will amplify inaccuracies when we move to multi-node systems.

3.6 Conclusion

We introduced two models to help developers gain insight into application speedup when a program runs on software disaggregated systems. We showed that applications with skewed system call distributions can tolerate higher forwarding overheads when a good forwarding policy is selected, but suffer from more dramatic effects when we forward the wrong system calls. To measure the effect of these overheads on real applications, we presented an open-source tool called `mktrace` which approximates forwarding overheads on existing systems. Using this tool, we demonstrated how to make speedup projections for applications without actually deploying them on a software disaggregated system. Finally, we validated our model using two real software disaggregated systems. Our model achieves 96.18% and 96.79% accuracy in these settings.

CHAPTER 4

TRACKFM: AUTOMATIC SUPPORT TO RUN APPLICATIONS ON SOFTWARE DISAGGREGATED SYSTEMS

Chapter 3 discusses the benefits of analytical modeling in understanding application speedups before developers port their applications to software disaggregated systems. In this chapter, we design, implement, and evaluate TrackFM,[16] a compiler and runtime framework that achieves full transparency (no porting effort) by recovering application semantics during compiler analysis. TrackFM leverages state-of-the-art compiler middle-end analyses and transformations to achieve high performance, using the heavily optimized AIFM runtime as a backend. No specialized hardware or modifications to the OS are required. Through a mix of micro and macro-benchmarks, we demonstrate that the compiler has sufficient knowledge to enable TrackFM to achieve near-performance parity (within 10%) compared to a library-based approach like AIFM, where developers manually modify code, while maintaining the programmer transparency of Fastswap

We summarize our contributions as follows:

- We introduce the compiler-based approach to software-based far memory, which provides a path to simultaneously achieving programmer transparency and performance on memory disaggregated systems.
- We demonstrate how to use modern compiler analysis and transformation techniques to automatically transform existing applications to support far memory.
- We introduce new compiler analysis and transformation passes that improve performance for the target applications.
- We present the design and implementation of TrackFM, a new compiler-based far memory system.

[16]The content of this chapter is based on published research [42]

Listing 4.1. Simple loop using AIFM's remote array.

```
int sum (RemoteArray * array, int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
      DerefScope scope;
      sum += array.at(scope, i);
  }
  return sum;
}
```

- We report on an extensive performance evaluation using numerous microbenchmarks and applications.

TrackFM is freely available online.[17]

4.1 TrackFM Design

Our goal is to use the compiler to approach the performance of library-based far memory solutions by automatically transforming existing applications, eliminating the need for programmer modifications. We aim to reuse the AIFM far memory runtime and automate its integration into the application. As an illustrative example, consider a `for` loop that computes the sum over an array of integers. To make this array remotable in the library-based solution (AIFM), the programmer must use the remote array type provided by AIFM libraries. The programmer must then change their code manually, as shown in Listing 4.1. The highlighted lines indicate programmer changes. Although these changes are minimal, they require understanding of AIFM's semantics; namely, a scope object must be provided so that AIFM does not evacuate in-use local memory. Moreover, modifying applications with large code bases to run on AIFM may not be practical.

We aim to transform unmodified C/C++ applications to use remote memory *au-*

[17]`https://github.com/compiler-disagg/TrackFM`

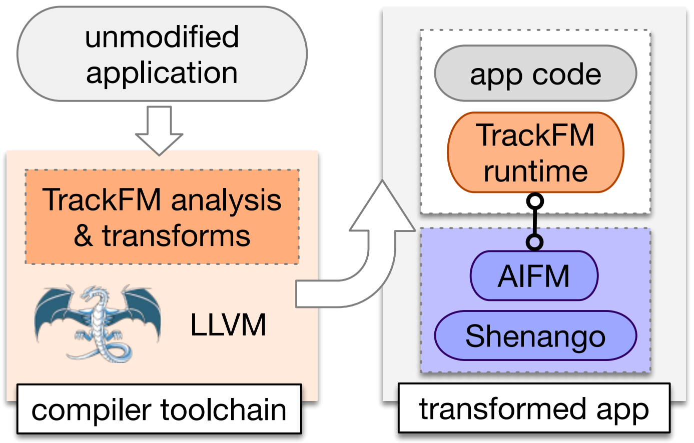


Figure 4.1. Users compile applications with TrackFM to run on a far memory cluster.

tomatically. Figure 4.1 shows our overall design. Our compiler toolchain takes the unmodified C/C++ source code[18] for an application, and using an LLVM-based, middle-end analysis and transformation pipeline, remotes certain memory allocations via AIFM. It also injects a thin runtime layer into the application that interfaces with AIFM. The toolchain produces a modified binary that runs on a far memory cluster. Our transformations take place at the IR level.

The primary obstacle to automating the integration with AIFM is the semantic gap between the application developer's high-level knowledge of data structures and what the compiler sees at the granularity of memory accesses. AIFM works at the level of *objects*, contiguous chunks of remotable memory, and what constitutes an object is determined by the application developer. For example, when AIFM's object size is set to 256B, a remote 1KB array will be represented by four chunked AIFM objects. A remote linked list, on the other hand, might use an AIFM object size of 64B to constitute a single linked list node. Unlike AIFM, TrackFM works on unmodified code, so it must automatically determine the mapping of memory allocations to AIFM objects using low-level information (i.e., by drawing boundaries around chunks of contiguous memory allocations). In kernel-based

[18]Our approach also applies for applications shipped as LLVM bitcode.

approaches, *any* page can be swapped to a remote node, while in AIFM, candidates for remoting are determined by which data structures the programmer uses the AIFM data types for. Our design strikes a middle ground, where any heap-allocated data can be swapped out (but not at the granularity of pages). Whether these heap-allocated regions actually *are* swapped out depends on temporal access patterns; hot regions will be kept local, while cold ones will be evacuated to the remote node. The TrackFM runtime tracks this "hotness" via AIFM's existing object access interposition mechanisms. AIFM has several programmer-directed parameters that affect its performance, for example, the degree of concurrency, object size, and prefetching strategy. We will see how the *compiler's* choices for these parameters impact performance in Section 4.3. Since our compiler framework requires source code, programs that use external libraries present a challenge. The naïve route is to ignore external libraries. Memory that they allocate will not be remotable. However, TrackFM needs to transform pointers to automatically remote them, and those transformed pointers can easily *escape* to library code, which does not know how to handle them. A library may then incorrectly attempt to access remote memory not yet localized by the TrackFM runtime. The alternatives are to (1) have programmers run external libraries through the TrackFM compiler or, (2) only allow pre-transformed versions of the libraries provided by us. We explore both options, though the latter is more pragmatic.

4.2 Implementation

We first outline how TrackFM transforms applications to use far memory, then we describe how we incorporate the high-performance AIFM runtime with TrackFM. Finally, we describe our compiler transformations in detail, including how we manage the overheads they introduce. We focus on realizing TrackFM in the context of C/C++ programs.

Far Memory Pointer Transformation The first distinction that TrackFM must make is between remotable and local-only pointers. AIFM makes this distinction using far mem-

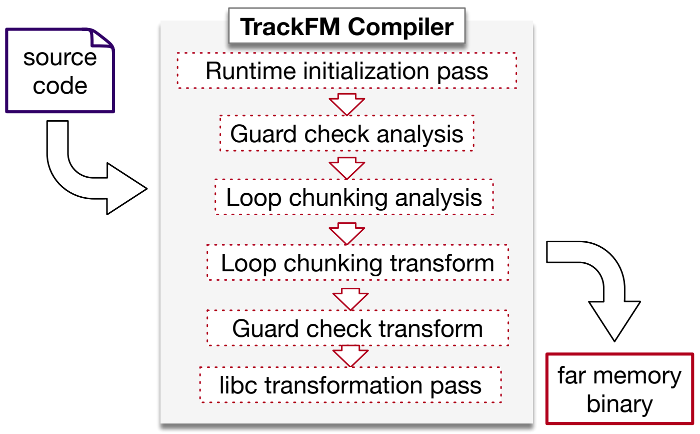


Figure 4.2. TrackFM's analysis and transformation pipeline.

ory data structures. However, TrackFM cannot rely on user annotations since we target unmodified code. Conceptually, all heap-allocated pointers must be managed by TrackFM, and all others (stack, global data, etc.) remain unchanged. However, as a pointer is just an address, we have no *a priori* way to tell them apart. TrackFM does this by overloading the higher-order bits of the address. In particular, it leverages x86 non-canonical addresses.[19] The 60^{th} bit of the address is used to flag a pointer as a TrackFM pointer. If this bit were to be set in any non-TrackFM pointer, the pointer would be invalid. To enforce this distinction, TrackFM provides a custom `malloc` implementation which replaces the default libc `malloc`. Our custom implementation always returns TrackFM (non-canonical) pointers.

Intuitively, a TrackFM pointer can refer to memory that is either on the local or remote system. Thus, the program must be prevented from using the pointer directly. The compiler must provide an indirection layer that, when the pointer is accessed at runtime, localizes the memory and produces a standard pointer in the local address space. Thus, we must *guard* accesses to TrackFM pointers. These guards constitute compiler-injected code that ensures memory is localized before access; they comprise the lion's share of

[19]Depending on the x86 implementation, the top 16 or 7 bits of a virtual or physical address must be either all zeros or all ones in order for the address to be "canonical." If a "non-canonical" address is ever used for an instruction fetch, a load, or a store, a general protection fault is triggered.

TrackFM's overheads, as we will see in Section 4.3. To properly guard pointers, the TrackFM compiler applies a series of analyses and transformations at the compiler's IR level (called passes), as shown in Figure 4.2. These passes are built on NOELLE [58], a novel analysis and transformation framework that expands LLVM [59] by introducing high-level and program-wide abstractions. We discuss each pass below.

Runtime Initialization To make far memory transparent to programmers, this pass inserts hooks in the program's `main` function to initialize TrackFM's runtime system.

Pointer guards In this pass, TrackFM searches for all LLVM IR-level load and store instructions that correspond to heap allocations (returned by `malloc`) and marks these instructions as eligible for guard transformation. The pass ignores accesses to stack and global objects by leveraging NOELLE's program dependence graph abstraction, which is powered by several high-accuracy memory alias analyses. Candidate heap pointers are later transformed by the guard transformation pass, described in Section 4.2.

Loop Chunking We introduce a novel loop chunking analysis to reduce guard overheads introduced in loop bodies. Our loop chunking pass incorporates NOELLE's profiling facilities when available to further improve our optimization. We describe the relevant transformation in Section 4.2, and techniques to improve it in Section 4.3.

Libc Transformation This pass transforms all memory allocation calls (mainly for heap allocation) in libc (e.g., `malloc`, `realloc`, `free`), into TrackFM-managed memory runtime calls. The TrackFM versions leverage AIFM's region-based allocator under the covers to allocate remotable memory. Custom heap allocators are not currently supported, but provided they simply replace libc malloc with their own managed heap, this would be trivial to add support for.

Bridging AIFM with the Compiler To integrate with AIFM, we use a lightly modified version of AIFM that includes hooks into the TrackFM runtime. We next discuss details about integrating TrackFM pointers with the AIFM runtime. In particular, we must transform contiguous heap allocations into AIFM *objects*, fixed-size chunks that can be either in the local or remote state. We will see in Section 4.2 that significant complexity arises because a given heap allocation can comprise multiple AIFM objects, each of which may be in different states (local or remote).

AIFM manages remotable memory at the level of individual data structures. Each of these data structures in the AIFM runtime is implemented as a C++ class which extends a base class that handles the underlying mechanisms of remote objects. We extend this base class with a unified *abstract* data structure (ADS) that the compiler uses to capture all remotable allocations for the application. With AIFM, programmers specify remote memory usage by leveraging one of these specialized data structures. However, with TrackFM, the compiler identifies *all* remotable allocations and attaches them to a single runtime-managed object pool. The ADS thus contains a pool of objects that represent the total far memory that an application can use. TrackFM interposes on an application's allocation sites and chunks the allocations into objects in the global pool at run-time.

Object size selection In AIFM, the user/data structure developer annotates each data structure with an object size for a given application. Since TrackFM does not require programmer changes, it is currently constrained to choose a single object size at compile time for the entire application. Unlike Fastswap, which is constrained by the page size, TrackFM supports object sizes smaller than a page, mitigating I/O amplification. While multiple object sizes are possible, this increases the complexity of the runtime system and compiler transformations, so we leave this for future work. We note that it is likely the case that only a few fixed object sizes make sense, and that these are likely to be powers of two ranging from 64B (cache line size) to 4KB (base page size). Using object sizes smaller than a cache line would saturate the network with many small packets, and would not take advantage of the network's bandwidth, which is geared to larger packets. On the other hand, much larger object sizes would suffer from I/O amplification, and defeat the purpose of sub-page granularity far memory. While the choice of object size is currently selected by us, the small search space suggests that an autotuning approach is feasible. Furthermore, if we are correct that only the powers of two from 6 (cache line) to 12 (base page size) need to be considered, an exhaustive search involving recompilation and a short-term execution would simply expand the short compile times.

Allocating far memory TrackFM only remotes heap allocations and maintains a simple non-canonical address space to service memory allocation calls by the application. All memory allocation call sites within libc are intercepted by TrackFM and will return TrackFM-managed pointers starting from the non-canonical address range (starting at address 2^{60}). Because TrackFM rewrites pointers at the middle-end, even if a pointer is cast to an integer type (for example to perform offset math), the resulting load/store will still be properly guarded, provided that the non-canonical bits of the address are preserved. Internally, TrackFM maps non-canonical pointers to objects in an ADS. The object corresponding to a TrackFM pointer can be derived by dividing the TrackFM pointer by the object size (a right shift for powers of two). A single memory allocation can span multiple objects, while smaller allocations are grouped into a single object.

TrackFM object state table Any particular allocation could be in a superposition, i.e., some of its constituent objects (chunks) could be local while others are remote.[20] AIFM tracks the local/remote state of objects by maintaining two metadata representations (one for each state) internally. Determining this state in AIFM requires two memory references,

[20]This is a property that makes compiler-based far memory different from prior DSM-focused systems.

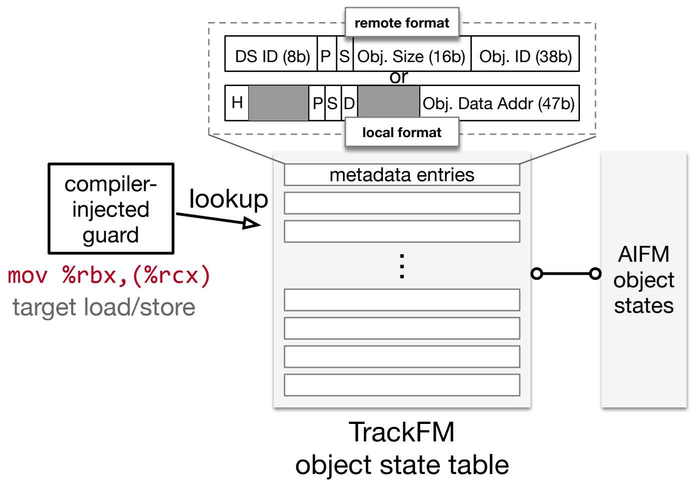


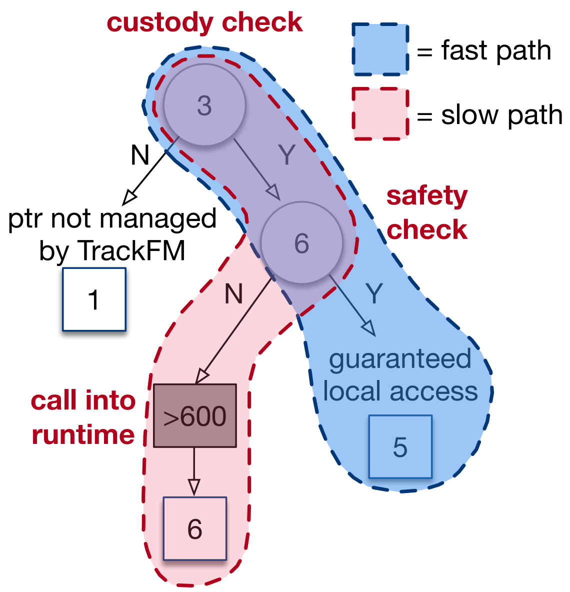
Figure 4.3. The object state table caches AIFM object metadata (shown on top, and reproduced from the AIFM paper [1]) for lighter-weight guards.

one to find the object, and another to access its metadata. TrackFM eliminates one of these operations by maintaining an *object state table*, an optimization that caches object metadata in a contiguous lookup table, allowing us to perform a simple index calculation rather than an indirect memory reference to derive object metadata. This is possible because of the way TrackFM encodes object IDs in the non-canonical range of the pointer. We modified AIFM so that this table is kept coherent with the AIFM-managed object metadata. The object state table contains metadata entries (8B each) for each object in the system, where the total number of objects is determined by the total size of the remote heap. The overhead of the table can be computed similarly to a single-level page table. For example, if we have a 32 GB remote heap (as in many of our experiments), we would need 2^{23} entries in the table (assuming each object is 4KB), thus remote heap (as in many of our experiments), we would need 2^{23} entries in the table (assuming each object is 4KB), thus consuming 64 MB for the full table. As shown in Figure 4.3, the compiler-inserted guard derives the object metadata from this table in order to determine whether or not the referenced object is localized.

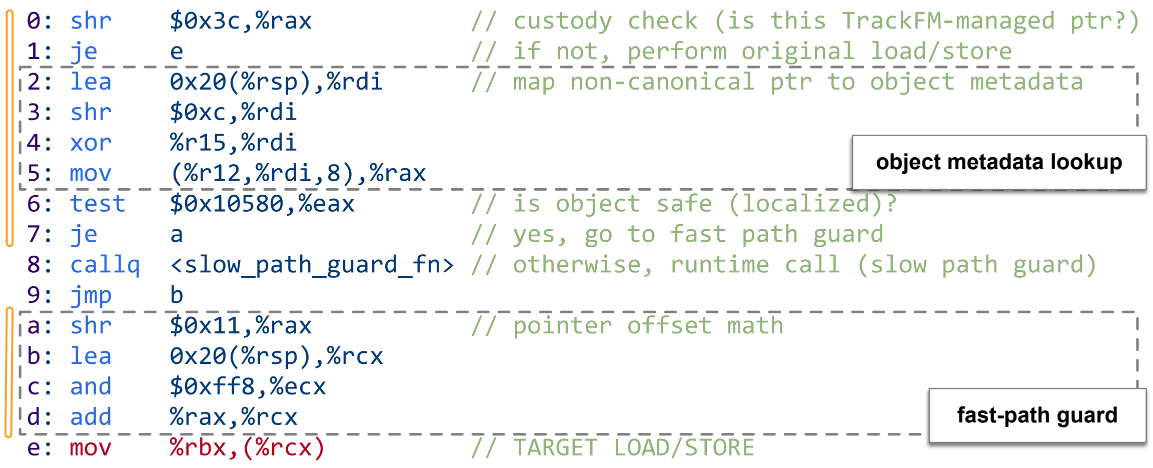
TrackFM Guards As described above, TrackFM instruments application derived LLVM bitcode with *guards* on every relevant load and store instruction referring to heap-allocated memory at the LLVM middle-end layer. These guards comprise compiler-injected instructions that ensure the memory is *localized* (brought into local memory) before being accessed. TrackFM guards localize an object by reverting the non-canonical address returned from the TrackFM allocator back into a canonical address before execution of the target load/store. Figure 4.4 depicts the guard. Figure 4.4(a) shows an abstract depiction of the injected code as a control flow graph, and Figure 4.4(b) shows the guard after it has been lowered to x86_64 assembly. We break down the TrackFM guard into three components: a *custody check*, a *fast-path guard*, and a *slow-path guard*. Note that on the fast path only one of those instructions is a data access (to the object state table) that can result in a cache miss. Figure 4.4(b) highlights the fast path through the guard with vertical orange lines on the left. Note that we can also enable optional debug instrumentation that indicates when guards take the fast or slow path, and which AIFM code path they trigger.

Custody check TrackFM first checks whether the pointer is managed by TrackFM. Recall that this means only heap-allocated memory. If a pointer is *not* managed by TrackFM, we immediately jump to the target load/store. This path constitutes roughly four instructions. If the pointer passes the custody check (i.e., it is a TrackFM pointer), we perform a table lookup to derive the object state table entry corresponding to the AIFM object, and then load the object state of the TrackFM pointer. This path constitutes roughly six instructions.

Fast-path guard We use AIFM's internal object metadata to determine if an object is *safe* to access, i.e., guaranteed to be local. Safety is satisfied if certain bits in AIFM's internal



(a) abstract control flow.



```
0: shr    $0x3c,%rax          // custody check (is this TrackFM-managed ptr?)
1: je     e                   // if not, perform original load/store
2: lea    0x20(%rsp),%rdi     // map non-canonical ptr to object metadata
3: shr    $0xc,%rdi
4: xor    %r15,%rdi
5: mov    (%r12,%rdi,8),%rax
6: test   $0x10580,%eax       // is object safe (localized)?
7: je     a                   // yes, go to fast path guard
8: callq  <slow_path_guard_fn> // otherwise, runtime call (slow path guard)
9: jmp    b
a: shr    $0x11,%rax          // pointer offset math
b: lea    0x20(%rsp),%rcx
c: and    $0xff8,%ecx
d: add    %rax,%rcx
e: mov    %rbx,(%rcx)         // TARGET LOAD/STORE
```

(b) generated x64 code.

Figure 4.4. **Left**: control flow of compiler-inserted guard check. Circles indicate conditional branches and squares indicate exit nodes. Each node is annotated with the number of x64 instructions executed. **Right**: guard lowered to x64 code. The vertical orange lines indicate the fast path (highlighted in blue on the left).

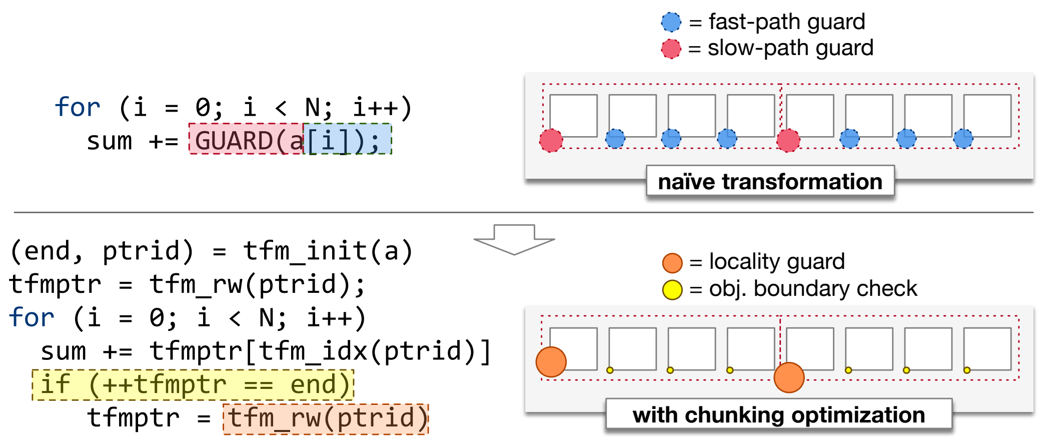
metadata representation are cleared.[21] When safety is satisfied, the fast-path guard will be taken, constituting 14 instructions. Note that it appears that there is a time-of-check to time-

[21]AIFM must indicate that the object has been localized either through a blocking access or an asynchronous prefetch request, and is not a candidate for evacuation to the remote node.

of-use issue between the `test` instruction (line 6) and the actual target load/store (line e). That is, if the safety check passes and this application thread gets context switched out (or even if there is a race), an evacuator might run on another core and delocalize the object, rendering the pointer invalid for the final target load/store. This issue is prevented because AIFM's evacuator threads use a barrier that waits on all application threads to converge to a state where remotable pointers are "out-of-scope." While within the context of a TrackFM guard, the app thread is guaranteed not to be in this "out-of-scope" state, preventing the convergence necessary for the evacuator to proceed. This means that between line 5 and line e, the object cannot be evacuated.

Slow-path guard If the object is *unsafe* to access, then we must call into the TrackFM runtime. TrackFM in turn calls into the AIFM runtime to dereference the object, which could involve a remote fetch. When TrackFM interfaces with AIFM here, it adheres to AIFM's internal `DerefScope` API (shown in Listing 4.1), and also triggers a periodic collection point to allow stale objects to be evacuated to the remote node. This runtime call in the slow path, which has a higher cost, ensures safety. Once TrackFM ensures safety, it performs the target load/store. The slow-path guard comprises at least 144 instructions when the pointer object is already localized. However, if the object is remote, the cost of the slow-path guard will be dwarfed by the remote fetch cost.

Managing Loop Overheads Up to this point, we focused on direct pointer accesses. However, there are many cases where pointers are accessed via an offset, a major example being array accesses. It is common for such accesses to occur in loops. Ideally, when iterating over a collection (e.g., an array) in a loop, we could localize the entire array at the beginning of the loop, bringing any remote elements local before accessing them. This optimization was commonly employed by compiler-based DSM frameworks [60, 61, 62]. However, because we build on AIFM, and a single collection might constitute *multiple*



```
for (i = 0; i < N; i++)
  sum += GUARD(a[i]);
```

```
(end, ptrid) = tfm_init(a)
tfmptr = tfm_rw(ptrid);
for (i = 0; i < N; i++)
  sum += tfmptr[tfm_idx(ptrid)]
  if (++tfmptr == end)
    tfmptr = tfm_rw(ptrid)
```

Figure 4.5. The loop chunking optimization eliminates fast-path guards within loops when object boundaries are not crossed. This trades off a cheaper conditional branch inserted in every iteration (yellow) and a more expensive guard at object boundaries (orange).

AIFM objects, the entire collection might be in a superposition (simultaneously local and remote). Moreover, the entire array may not fit in memory. This renders the DSM-style hoisting optimizations ineffective, and it means that all pointer accesses *within a loop body* must be guarded.

However, when many collection (array) elements fit within a single AIFM object, many of these guards are redundant. They are only necessary when we cross object boundaries in the loop. In AIFM, the iterator classes developed by the library developer for the remote data structures manage this overhead. With TrackFM we leverage the compiler's knowledge of the loop to reduce this overhead by developing a *loop chunking optimization* for TrackFM pointers.

Figure 4.5 depicts such a situation with a contiguous array, where multiple array elements fit within an AIFM object. The naïve guard insertion strategy will involve injecting guards at every element access. The slow-path guards (shown in red) will be taken at object boundaries, i.e., when `i` is a multiple of the object size, and fast-path guards (blue) will be taken on every other access. With our optimization, the compiler can determine the induction variable of a loop, including the step count and the start value of the induction variable, so it knows that sequential accesses within the boundaries of an already fetched object do not require fast-path guards. This trades off many fast-path guards with a slightly more expensive *locality invariant guard* at object boundaries that calls into the runtime to pin the object in local memory for the duration of accesses to the object (one loop chunk). *Object boundary checks* (yellow) are also inserted on every access to detect when the locality invariant guard should be taken. Note that this optimization is not just applicable to contiguous arrays; it applies more generally to loops that employ a loop-governing induction variable, which is common in practice (we will see this in Section 4.3). The analysis pass for the loop chunking optimization searches for spatially local memory accesses that occur in loops, typically a popular location of hot code. Upon finding these accesses, TrackFM attempts to mitigate the overhead from guards in the loop body by chunking the original pointer into object-size chunks. To identify such memory accesses, TrackFM makes use of NOELLE's induction variable (IV) analysis.[22] Such analysis is unique as it detects induction variables as patterns in the dependence graph, rather than building on variable analysis. This leads us to capture significantly (∼3×) more induction variables than what is traditionally possible. However, TrackFM can also be adapted to use other IV analyses should better techniques arise. Note that there is not a correctness issue if the IV analysis misses induction variables; it just results in lost loop chunking optimizations. We plan to further generalize our loop analysis in the future, for example by adapting polyhedral methods [63] to NOELLE. Our optimization is particularly effective for workloads that display high regularity.[23] Prefetching plays an important role in such workloads. TrackFM can detect sequential access at compile-time, so it uses prefetching alongside loop chunking to mitigate loop overheads. This has an increasing impact on performance as the number of

[22]This is a more sophisticated analysis than what is available in gcc or LLVM. See §2.B and §4.C of the NOELLE paper [58] for details.

[23]That is, spatial locality of access closely matches temporal locality of access.

pointers iterating over induction variables in a loop increases. This demonstrates a strength of the compiler-based approach to far memory: kernel-based approaches cannot take advantage of such loop-centric memory analysis; they must make *post hoc* inferences based on run-time page faults.

Improving Loop Chunking Loop chunking is not *always* beneficial. In particular, when array elements are large (so that fewer of them fit within a fixed-sized AIFM object), or the loops have a small iteration space, there are fewer fast-path guards in the first place. If we apply the loop chunking transformation in such cases, performance can actually drop relative to the standard guards. Intuitively, there is a break-even point when sequential array access occurs at a fine enough granularity for this transformation to pay off. To help the compiler determine where that point is, we develop a simple cost model.

Cost Model Let o be the size in bytes of a TrackFM object, and let e be the size of an element in a collection accessed in a loop. For example, for an 8-byte integer, e would be 8. We model the number of elements that fit within a single TrackFM object as the *object density*, $d = \frac{o}{e}$. We are interested in determining how densely elements must be packed before the compiler applies the loop chunking transformation. Intuitively, the more dense an object, the more fast path guards will be involved, so the more advantageous the optimization will be. Conversely, if there are few elements per object, the transformation could be detrimental. With the naïve transformation, each loop will iterate over some number of objects, and each object must incur a fast-path guard for each element access, except for the first, which requires a slow-path guard. For each object there will thus be one slow-path guard and $d - 1$ fast-path guards. Slow-path guards have cost c_s and fast-path guards have cost c_f. We model the guard costs at the level of individual objects. We can then estimate the cost of the entire loop in terms of guards:

$$C = (d-1)c_f + cs$$

Our loop chunking optimization replaces fast path guards (14 instructions) with less expensive *object boundary checks* (3 instructions) that determine when an object boundary is crossed. The object boundary checks are shown as small, yellow circles in Figure 4.5. Slow-path guards are replaced with slightly more expensive *locality invariant guards* (orange circles) at object crossing boundaries, which involve a call to the runtime. We model the cost of the boundary checks as c_b and the locality invariant guards as c_l. The cost of the transformed loop in terms of guards is then:

$$C_{\text{opt}} = (d-1)c_b + c_l$$

When a loop iterates over large elements, the relatively high cost of the invariant guard can offset the elimination of the fast-path guards. Thus, we must only apply the optimization when there is sufficient object density, i.e.:

$$d > \frac{c_s - c_l}{c_b - c_f}$$

Figure 4.6 shows the projected cost of a simple loop with a varying number of iterations for the baseline method and the loop chunking optimization. The chunking optimization becomes preferable once an object comprises as few as ~730 elements. The curve on the plot shows empirical measurements of loop cost. Note that the projected break-even point matches the empirical data. Thus, if the compiler can determine d, we can make intelligent choices about when to apply the loop chunking transformation. To do this, we leverage NOELLE's profiling engine to collect loop code coverage statistics. With the profiling pass in TrackFM we filter out loops with low object density transparently without

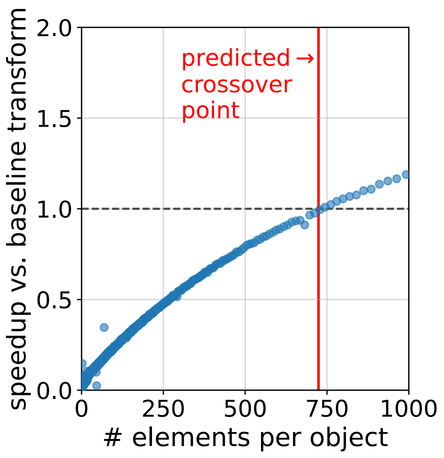


Figure 4.6. Cost model to capture the point at which loop chunking becomes advantageous. The horizontal dotted line shows empirically when loop chunking benefits, and the vertical red line shows when the model predicts a beneficial outcome.

modifications to source code.

4.3 Evaluation

The TrackFM compiler must make choices about how it structures far memory objects and passes information to the runtime system. We evaluate the performance impact of these choices and the overheads of compiler-injected guards using microbenchmarks, studying the impact of different types of workloads and access patterns in a controlled setting. We then demonstrate that by making good choices, TrackFM can approach the performance of AIFM on application benchmarks while maintaining programmer transparency. We seek to answer the following questions in our evaluation:

- How expensive are TrackFM guards? (§4.3)
- To what degree can compiler analysis and transformations mitigate guard overheads? (§4.3)
- When the compiler can control AIFM object size and prefetching, how do its choices impact performance? (§4.3)

- To what degree can TrackFM mitigate I/O amplification? (§4.3)
- How do TrackFM's optimizations translate to overall application performance relative to state-of-the-art approaches? (§4.3)
- How does TrackFM affect code size and compilation time? (§4.3)

Experimental setup We conducted our experiments on CloudLab [64] using two x170 machines with 10-core Intel Xeon E5-2640v4 CPUs clocked at 2.40 GHz, 64GB RAM and a 25 Gb/s Mellanox ConnectX-4 NIC. We used Ubuntu 18.04 (to support AIFM) with stock Linux kernel version 5.0 and DPDK version 18.11. For Fastswap measurements we use the latest version[24] ported to the 5.0 kernel.[25] We use the most recent publicly available version of AIFM.[26] TrackFM builds on LLVM version 9.0.0, with NOELLE v9.8.0. For C++ applications, we use libc++ version 9 provided by clang (we directly compile it with TrackFM). For large codebases we use WLLVM[27] to produce bitcode for the entire application before passing it to the TrackFM compiler.

Guard Overheads The primary source of TrackFM's overhead comes from the compiler-inserted guard instructions at the bitcode level on heap-allocated loads and stores. Table 4.1 shows their costs in cycles relative to local load/store operations. The additional overhead for a fast path guard relative to a local unmodified load/store (36 cycles) instruction is 21 cycles. This will be the common case for applications that have locality of access. The uncached slow-path and fast-path guards are more expensive, but better than a page fault.

[24]commit 9cfc2a

[25]`https://github.com/nilyibo/fastswap`, commit 9d5c6f

[26]`https://github.com/AIFM-sys/AIFM`, commit aaf711

[27]`https://github.com/travitch/whole-program-llvm`

Table 4.1. TrackFM fast-path vs. slow-path guard costs when a object is local. Costs are reported in median cycles over 1000 trials.

TrackFM Guard Type	Cached	Uncached
TrackFM fast-path read guard	21	297
TrackFM fast-path write guard	21	309
TrackFM slow-path read guard	144	453
TrackFM slow-path write guard	159	432

Table 4.2. Comparison of primitive overheads for TrackFM and Fastswap. Costs are reported in median cycles over 1000 trials.

Runtime Event	Local Cost	Remote Cost
Fastswap read fault	1.3K	34K
Fastswap write fault	1.3K	35K
TrackFM slow-path read guard	453	35K
TrackFM slow-path write guard	432	35K

The slow-path guard is similar in cost to a major page fault in Fastswap when an object is not present in local memory because both events trigger a remote fetch over the network. For reference, Table 4.2 compares slow-path guards to remote page fault costs in Fastswap (both when the page is local and remote). Handling a page fault in the kernel incurs 2.9× the cost of handling a slow-path guard in TrackFM when the data is local. This changes when the object/page is remote due to Fastswap's fast RDMA backend, which outperforms our use of AIFM's TCP-based backend (from Shenango) when there is not sufficient concurrency. However, even with this high-performance networking layer, Fastswap still provides little benefit over our remote slow-path guard. This is due to Fastswap's page fault handling overheads (e.g., mapping and cgroups memory reclamation). If we really instrument every load and store to heap-allocated memory, what would the costs be? To provide initial intuition, we used TrackFM to automatically transform the STREAM benchmark [65], which has a 9GB working set. This transformation produces up to 56 million slow-path guards and $\sim$10 *billion* fast-path guards. Note that we must pay the cost of these guards *even when objects are local*. Neither kernel-based approaches nor library-based approaches pay such costs for local objects (though AIFM does incur overhead for smart pointer indirection). Thus, it would seem that these guards present an insurmountable barrier to achieving good performance. However, as we will see in the next Section, TrackFM can exploit regularity in the workload to dramatically reduce the number of guards.

Mitigating Guard Costs To make compiler-assisted far memory feasible, there are two paths to increase performance: (1) reduce guard costs and, (2) reduce the *number* of guards. We spent significant effort on (1), making the common case fast-path guard involve a small number of instructions (only 14). In this Section, we focus our discussion on the second path.

Loop chunking transformation Loop chunking, described in Section 4.1, eliminates fast-path guards, a key factor for improving performance. To understand its impact, we first evaluate its effects on the STREAM benchmark, which involves sequential access to arrays of small elements (integers), and is simple to transform. The "Sum" test consists of a single memory access to an array element (`sum+=a2[i]`) within a loop. "Copy" consists of two memory accesses (`a1[i]=a2[i]`) within the loop body. Figure 4.7 shows the speedup when using the optimized loop chunking transformation relative to the naïve transformation, where every loop element involves a fast-path guard. The total working set size for both examples is fixed at 12GB to aid in comparison. Note that the local memory constraint enforced on the application does not include the metadata used by AIFM/TrackFM.

We see that as the number of memory accesses within the loop increases (looking at the figures top to bottom), the speedup offered by loop chunking increases due to the larger

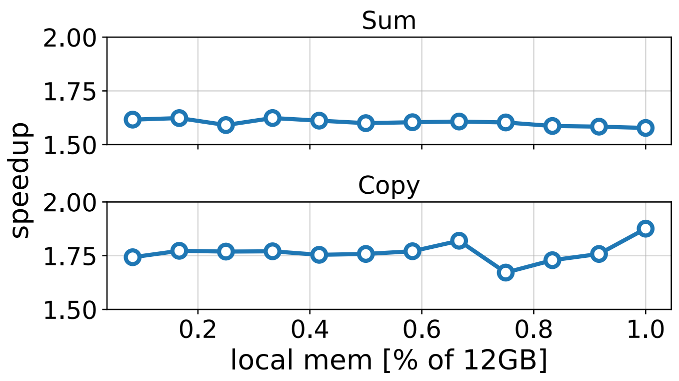


Figure 4.7. Speedup from loop chunking improves with increased memory accesses in loops as more fast-path guards are eliminated.

number of fast-path guards that are eliminated. For example, for "Sum," we reduce the fast-path guard count from ~1.6 billion to *zero*. Notice that the horizontal axis sweeps the amount of local memory available to the application, with increasing memory pressure to the left. These graphs tend to have an inclination towards the right-hand side since in that regime the system is less network-bound, so the importance of eliminating guard overheads is amplified.

Improved Loop Chunking To showcase how profiling can be coupled with our cost model from Section 4.2, we automatically transformed a *k-means* benchmark, which contains many loops for which it would be detrimental to apply the loop chunking transformation. We run *k-means* with 30 million points. The working set size is fixed at 1GB.

Figure 4.8 shows the results of applying the loop chunking optimization indiscriminately to *all* loops compared to applying it only to those loops identified as viable candidates by the TrackFM profiler, according to our cost model.

Both lines are normalized to the baseline (no loop chunking) to measure speedup. The figure shows that applying the loop chunking transformation indiscriminately produces poor results and suffers on average 4× slowdown. This is because *k-means* has many nested

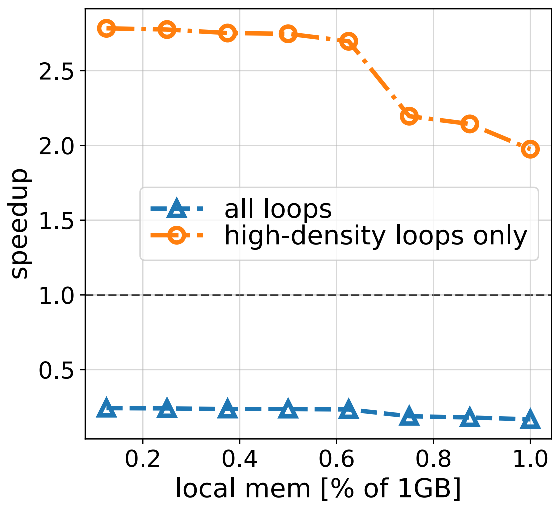


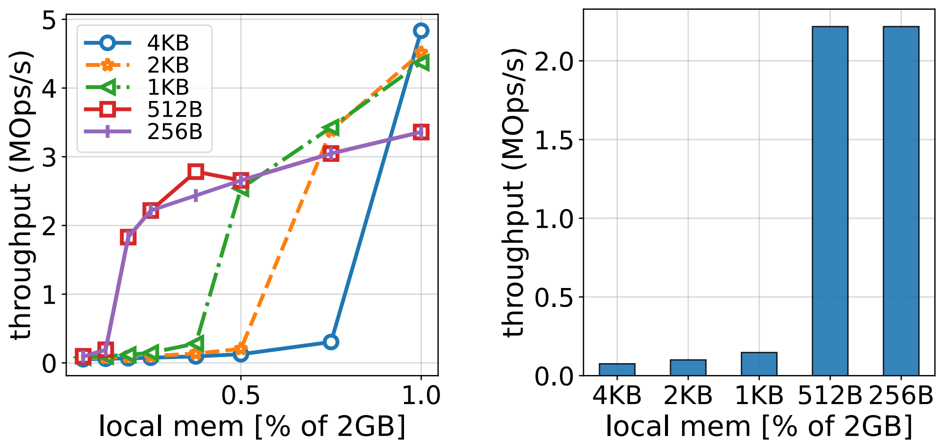
Figure 4.8. TrackFM can selectively apply the loop chunking optimization (like in *k-means*) to avoid collections with low object density.

loops with a low object density. Such nested loops amplify the cost of loop chunking. In this case, there were at least 512 array elements per AIFM object. The chunking optimization detects 103 array pointers, and after applying the cost model only 27 were optimized. Applying the cost model to the loop chunking pass here improves the situation considerably, resulting in a mean speedup of 2.5×.

AIFM Parameters The TrackFM compiler must make two primary choices when integrating with AIFM: the object size and prefetching strategy. This Section explores the impact of those choices.

Object size TrackFM currently chooses an object size at compile time, though this choice could in principle be informed by profiling. To evaluate the impact of this choice, we compare two microbenchmarks with different degrees of spatial locality and granularity of access.

The first microbenchmark involves accessing a hashmap, much like how a key-

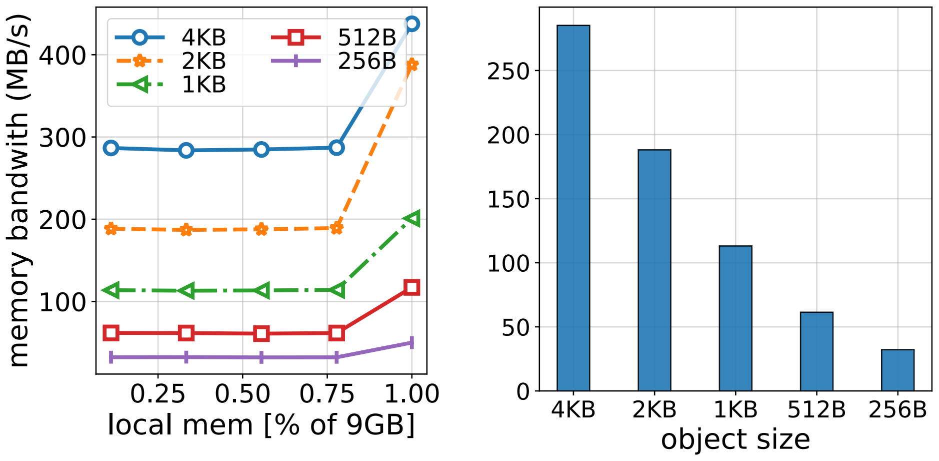


(a) (b)

Figure 4.9. Impact of object size on STL maps. Fine grained memory accesses with little spatial locality can benefit from small object sizes.

value store would operate. We use the unordered hashmap implementation from the C++ STL. Both keys and values are 4B integers. In this case, the entire C++ STL is transformed by the TrackFM compiler. The working set size is 2GB. We use a workload generator to access the hashmap (50 million lookups) according to a Zipfian distribution with skew 1.02. To generate the access trace, we store a sequence of keys sampled from the distribution in a separate 190 MB array also allocated on the heap. In this case, a small handful of the entries in the hashmap will constitute the majority of accesses, so there will be a high degree of temporal locality (but little spatial locality), and accesses occur at very small granularities (4B). The left side (Figure 4.9(a)) shows the impact of varying object size as we sweep the amount of local memory available, and the right side (Figure 4.9(b)) highlights the impact for a fixed proportion of the working set size available to local memory (25%). We measure the throughput (MOps/s) of the generated workload. In this case, a smaller object size is clearly preferable.

If we look again at STREAM, where the access pattern shows almost perfect spatial locality, we would expect to see different results. Here, we use the "copy" benchmark from STREAM with a working set size of 9GB. In this case, we measure the far memory



(a) (b)

Figure 4.10. Impact of object size on STREAM. Access patterns with high spatial locality benefit from the choice of a larger object size.

bandwidth (the default metric reported by STREAM). Though the granularity of access for this example is even smaller (integers), the high degree of spatial locality necessitates chunking elements into larger objects. In this case, 4KB is the better choice.

Figures 4.9 and 4.10 highlight that proper selection of object size is critical to performance. While we currently make this choice offline, we envision using profiling to make this choice when application code is recompiled with TrackFM.

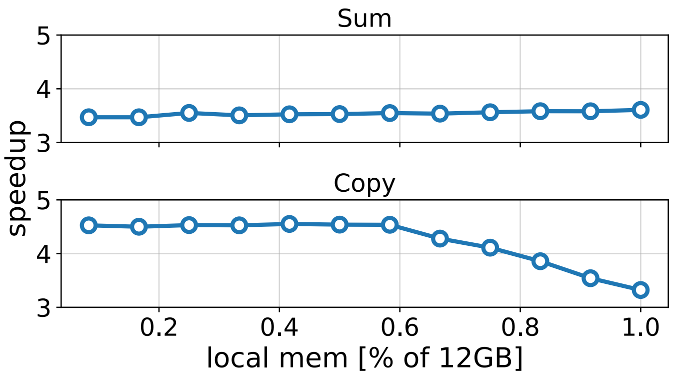


Figure 4.11. Speedup of prefetching coupled with loop chunking vs. only loop chunking. The combinations helps TrackFM extract more performance from workloads with spatial locality.

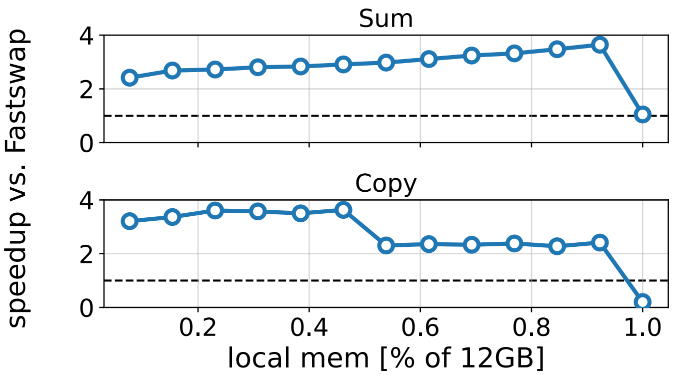


Figure 4.12. Speedup on STREAM relative to Fastswap with prefetching and loop chunking enabled. TrackFM's memory analysis helps to best exploit AIFM's high-performance prefetching.

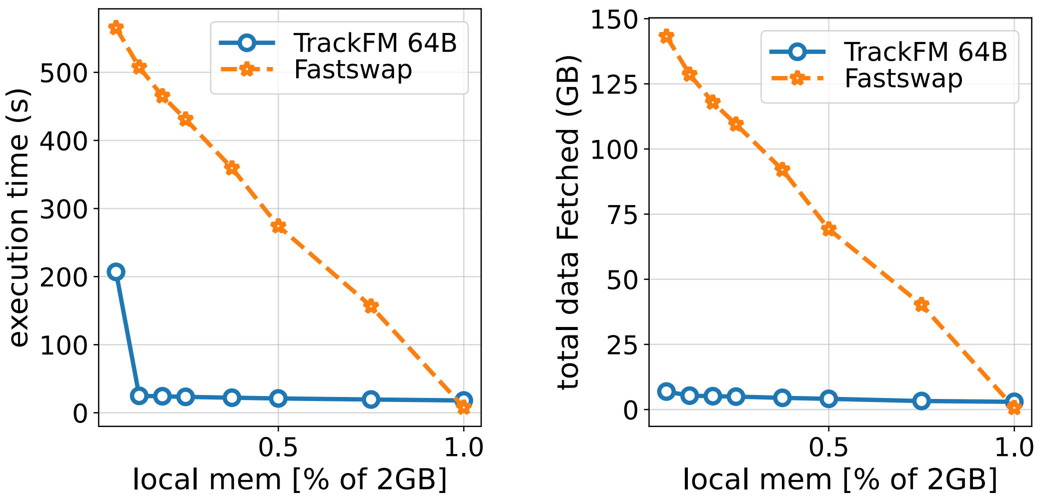
Prefetching When much of the application's memory is remote, the costs of remote fetches can dominate execution time. To mitigate network costs in this regime, TrackFM must employ prefetching to exploit spatial locality. We again run an experiment on STREAM, this time with and without prefetching enabled. In this case, we use AIFM's existing stride prefetcher, and we prefetch pointers operating on induction variables as identified by TrackFM's loop chunking pass. Figure 4.11 shows the speedup of using prefetching relative to no prefetching as we sweep the amount of local memory available. The loop chunking optimization discussed previously is enabled in both cases. If we focus on the left-hand side of the figures (where remote costs dominate), we see a large impact (almost 5×) on overall performance. As more local memory is available, the cost of guards dominate, so the impact of prefetching reduces. We validated this with experiments (not shown for space) that demonstrate that the relative number of *critical* remote fetches, i.e., the number of loads/stores blocked by first having to fetch the object from remote memory when prefetching is disabled, is reduced dramatically with prefetching.

Figure 4.12 shows the speedup relative to Fastswap on STREAM when we apply both chunking and prefetching. TrackFM performs ~2.7× better than Fastswap for Sum, and ~2.9× better for Copy. In this case, Fastswap is limited by its page fault costs, and by its weaker ability to discern high-level knowledge about the access pattern. Note that AIFM could achieve similar (even slightly better) performance here, but would require programmer modifications.

Mitigating I/O Amplification One of AIFM's major goals is to reduce I/O amplification, i.e., the unnecessary localization of unused memory, for workloads that access memory at fine granularity. Can TrackFM achieve the same goal? Figure 4.13(a) recreates our hashmap example, which will be sensitive to I/O amplification due to the small key/value pair sizes (4B). This time we show how overall performance is highly correlated with the amount of data transferred. We see how the smaller object size chosen by TrackFM significantly reduces the amount of data transferred over the network relative to Fastswap, which uses the standard 4KB page size. Fastswap transfers 43× the working set size for the hashmap, while TrackFM amplifies the working set by only 2.3× (the 64B object size chosen here is still larger than the key/value pairs). The net effect of reducing I/O amplification in this case is an average speedup of 12× relative to Fastswap. Though AIFM can achieve similar or higher speedups with programmer effort, this involves porting libc++ (457 KLOC) to AIFM, a non-trivial task.

Just the array storing the access trace for the keys requires 190MB, and with local memory constrained to 5% of total application memory (only 128 MB), we see high memory pressure, resulting in many object evacuations and swap-ins. Thus, we see an inflated execution time (~200s) for the first point to the left of Figure 4.13(a).

Application Benchmarks How do injected guards, remote costs, and our optimizations translate to overall application performance? We explore this question with two application



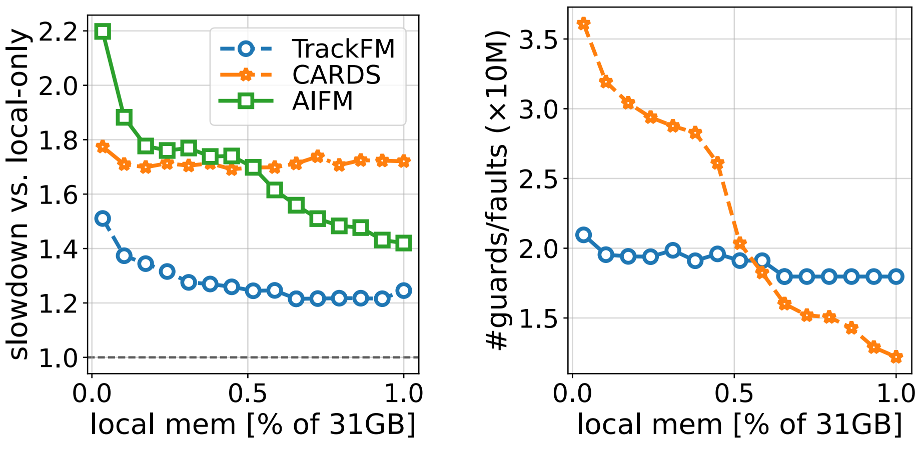
(a) (b)

Figure 4.13. Applications that access memory at small granularities suffer when limited by the architected page size.

benchmarks. The first is a data analytics workload taken from Kaggle [28] that analyzes New York City taxi trips. We adapted this benchmark from AIFM to validate our results against that paper [1]. The second application is memcached [66], a commonly used in-memory key-value store. We also evaluate several benchmarks from the widely used NAS suite [67].

Analytics Application The analytics application has a working set size of 31 GB. We compare the performance of the application automatically transformed with TrackFM to the same application running on Fastswap and AIFM. This analytics application builds on a custom C++ dataframe library, and while we can correctly transform this library, our loop optimizations will not work efficiently due to C++ semantics such as exception handling, which the existing loop analysis in NOELLE has no support for. We concluded that supporting/extending NOELLE to support such C++-specific semantics would require engineering effort not justified by the research value added. Instead, we ported the original

[28]`https://www.kaggle.com/code/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook`



(a) (b)

Figure 4.14. Performance of analytics application on TrackFM vs. Fastswap and AIFM. The left (a) shows overall performance normalized to a setup with only local memory, varying the amount of local memory available to the application. The right (b) shows the number of guard checks for TrackFM and page fault events for Fastswap. With less local memory, the page fault cost for Fastswap is amplified.

C++ dataframe library used in that paper to C, and the results reported for the analytics workload use the C dataframe library.

Figure 4.14 shows that when the available local memory is constrained, TrackFM comes within 10% of AIFM's performance, reaching near parity. Fastswap's performance converges when remote costs stop dominating, when roughly 75% of the working set fits in local memory. To explain these results, we measured the number of major (remote) page faults in Fastswap and the number of slow-path guards injected by TrackFM. We see that the page fault count (page faults imply one-sided RDMA operations in Fastswap) is relatively much higher than TrackFM guards; both event counts strongly correlate with overall performance. The analytics application consist of many column scan operations, which involve tight loops with almost no temporal locality but a high degree of spatial locality. TrackFM can exploit this to eliminate much of the guard costs, and also benefits from the AIFM runtime. How impactful is our loop chunking optimization here? Figure 4.15 breaks down the performance loop chunking, with loop chunking applied to all loops, and with it

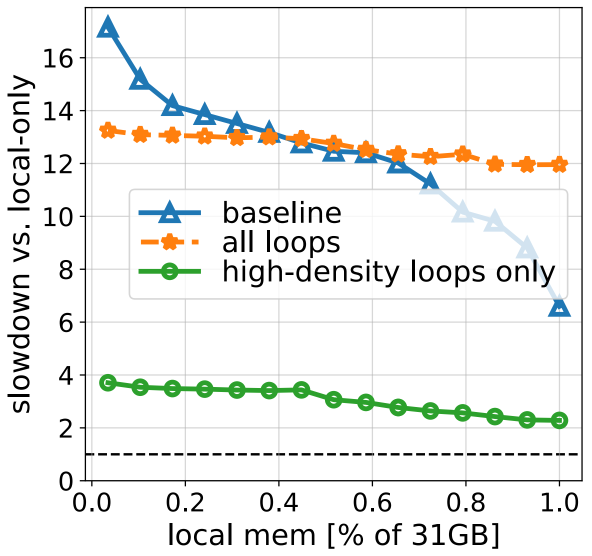
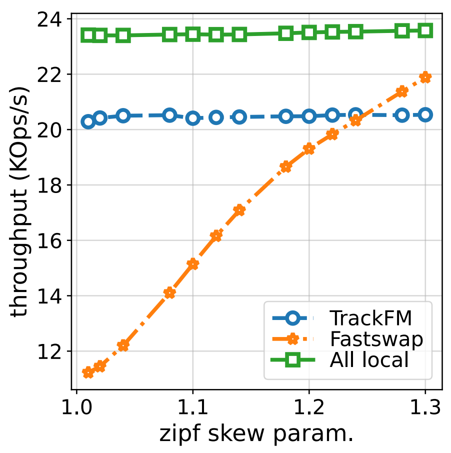


Figure 4.15. Applying the loop chunking optimization to low density objects in the analytics application reduces performance.

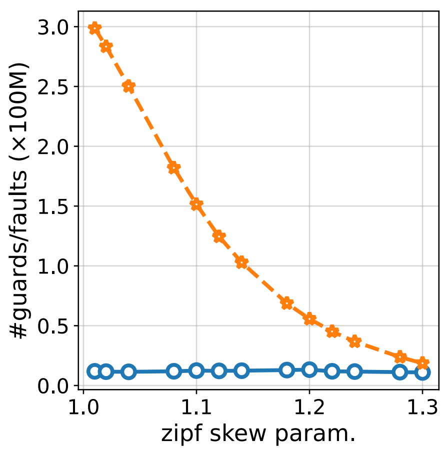
applied only to candidate loops identified by our cost model. This application has several aggregation operations that involve loops that iterate over small collections of table rows (low object density), so applying the model here clearly has benefits for reducing guard costs.

Memcached In-memory key-value stores represent another end of the access pattern spectrum. Here, access patterns tend to show much less spatial locality, and the granularity of access tends to be quite small, thus there is significant sensitivity to I/O amplification. We use TrackFM to automatically transform memcached version 1.2.7 to run as a far memory application.

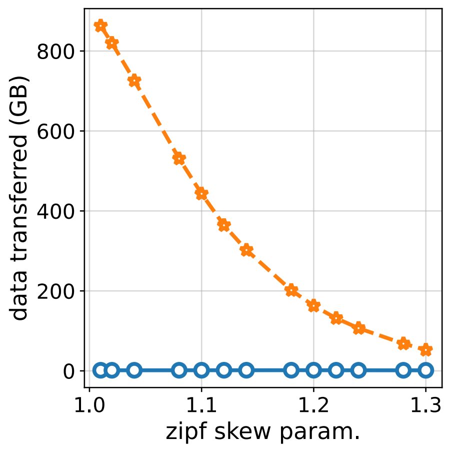
We use key/value pair sizes based on the USR distribution [68]. The working set size for memcached is 12GB, and we constrain the local memory to 1GB. We use a workload generator to create *get* operations on a Zipfian-distributed set of 100M keys. We measure the overall throughput for all get operations. Figure 4.16 shows the results. TrackFM shows a ~1.7× improvement over Fastswap when the skew parameter for the access distri-



(a)



(b)



(c)

Figure 4.16. Key-value stores with small object sizes and little spatial locality suffer from I/O amplification in Fastswap.

bution is between 1.01 and 1.04. As the access distribution becomes more skewed, we see an average speedup of 1.3× over Fastswap. As the skew parameter increases, Fastswap's performance converges due to increased temporal locality, which helps to amortize its page fault costs. While not shown in the figure, as we increase the amount of memory on the local node, Fastswap will converge with an even smaller skew, since more hot keys in the working set can fit on the local node. In this regime, TrackFM's fast-path guards become expensive, as they are not amortized like page faults. As the access distribution becomes less skewed, however, TrackFM outperforms Fastswap due to reductions in I/O amplifica-

Table 4.3. NAS benchmarks (C++ versions) run on TrackFM.

Benchmark	**Class**	**Memory (GB)**	**LoC**
CG (conjugate gradient)	D	9	586
FT (3D FFT)	C	6	756
IS (bucket sort for integers)	D	34	558
MG (PDE solver with multigrid method)	D	27	941
SP (PDE solver with scalar penta-diagonal method)	D	12	2013

tion. We verify this by measuring the total data transferred over the network. Figure 4.16(c) shows that Fastswap, limited by the architected page size, transfers 66× the working set size, much of which is unnecessary since the key-value pair sizes are small. In contrast, TrackFM benefits from small object sizes and transfers only 15× the working set.

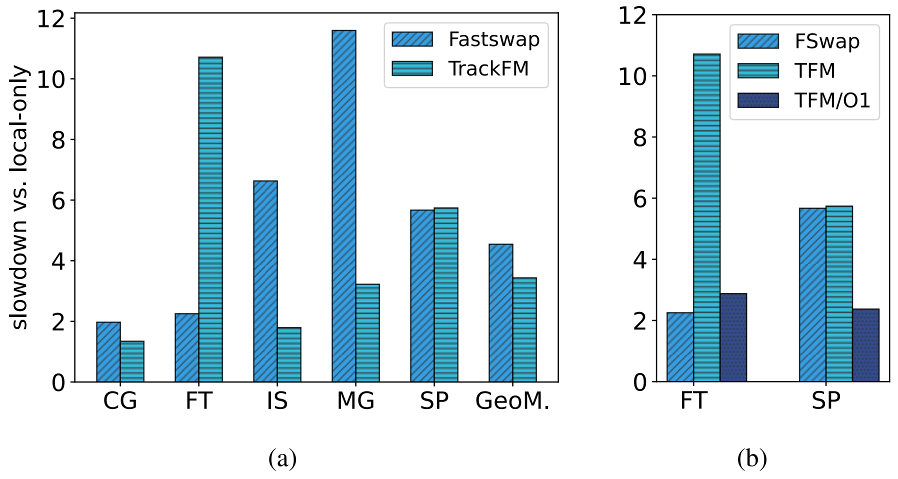


Figure 4.17. NAS benchmarks configured with a local memory size of 25% of the application working memory. Performance is normalized to local memory.

NAS benchmarks We use a reference C++ implementation of the NAS serial benchmark suite [69], and select a limited subset (details shown in Table 4.3) due to time constraints. Figure 4.17(a) shows TrackFM outperforming Fastswap for most benchmarks, where page faults are the limiting factor for Fastswap. FT is a notable outlier where TrackFM performs poorly. First, the FFT implementation in NAS has a particularly friendly access pattern for Fastswap involving good temporal reuse, allowing it to amortize its page fault costs. Further investigation revealed that TrackFM is also injecting an exceptionally large number of guards for FT. We found that the deeply nested, tight loop structure used in FT confounds our loop analysis, resulting in the high guard count. However, we found that this is mainly an artifact of the default analysis pipeline in NOELLE. By default, NOELLE sees unoptimized code from LLVM. However, in our case, it makes more sense to accept pre-optimized code in NOELLE to minimize the number of guards that are injected. For example, redundant code elimination or dead code elimination can reduce the number of loads and stores and thus the number of guards. We verified this in Figure 4.17(b), where we perform the chain of optimizations included in the "O1" set *before* the TrackFM passes (TFM/O1). This results in a 6× reduction in memory instructions for FT, and a 4× reduction for SP, dramatically reducing guard overheads. This experiment led us to change NOELLE's default optimization pipeline order for use with TrackFM.

Compilation Costs TrackFM increases generated code size by an average of 2.4× relative to the original binary. This increase is roughly proportional to the number of memory instructions in the program, each of which is expanded into a guard with the standard transformation. TrackFM's compile time is under 6× compared to standard LLVM, though we have not yet focused effort on reducing compilation overheads.

4.4 Discussion

Section 4.3 showed that the *compiler-based* approach holds promise. We now attempt to convey some hard-earned insights from our work, its limitations, and future prospects.

Lessons We spent significant effort engineering the guards to be lightweight. This did pay off, but we were surprised to find that exploring ways to *eliminate* guards entirely was the more fruitful path, though this is somewhat obvious in retrospect. We were also surprised how well kernel-based approaches perform when there is sufficient *temporal* locality. This is because page fault costs are quickly amortized when there is repeated access. Even in this scenario, however, they are still sensitive to I/O amplification. This suggests that a hybrid approach (compiler and kernel) holds promise.

Understanding the high-level semantics of access patterns (i.e., access over an array, or a list, etc.) is critical for performance. We expect greater benefits when we can capture information about recursive data structures [70]. Finally, we found that in some cases, application code optimized for locality of reference can actually confound efforts by the compiler to derive fine-grained information about the access pattern. For example, memcached uses an optimized slab allocator that batches small allocations, thus grouping together small objects into large chunks. This actually limited TrackFM's ability to mitigate I/O amplification; TrackFM could have more effectively transformed this application had it performed small allocations in the naïve way.

Hardware Support The overhead of TrackFM's guards could be improved with new hardware extensions. In the limit, the hardware can interpose on remote accesses and track dirty objects on its own, for example by extending the cache coherence engine (as in Kona [71]). However, while this approach is attractive from the standpoint of transparency, it forgoes the benefits of the high-level knowledge available to the compiler. An extension more appropriate for TrackFM might involve hardware that the compiler could manage, e.g., a lightweight, sub-page triggering mechanism that vectors *directly* to user-space (in contrast to the existing userfaultfd mechanisms in Linux [72]). This might, for example, look like a software/hardware stack built atop range translations [73] and user-level fault handling.[29]

[29] As in Intel's user-level interrupt vectoring introduced in the Sapphire Rapids microarchitecture [74]

Limitations and Future Work The impact of AIFM's object size parameter is workload-dependent, so users must currently choose it. We believe it would be fairly simple to remove this engineering limitation by using autotuning, as discussed in Section 4.2. Since TrackFM operates at the level of LLVM IR, information about application semantics (e.g., recursive data structures) is mostly lost. We plan to explore inter-procedural data structure analysis [75] to capture these semantics. There is also opportunity for languages whose memory semantics more closely match those of far memory, such as Rust, whose ownership model maps well to the notion of locality. High-level parallel languages, where ownership can fall out of language semantics [76], and partitioned global address space (PGAS) languages [77] could also map to compiler-based far memory.

Fetching remote data just to perform trivial computations is unwise. AIFM overcomes this by allowing library developers to manually offload such lightweight computations onto the remote node, thus employing *near-data processing*. We believe TrackFM could employ static analysis techniques, such as automated amortized resource analysis [78, 79], to achieve the same goal. TrackFM could also benefit from a profiling stage that prunes the set of heap allocations available for remoting based on access frequency. For example, the MaPHeA framework leverages hardware performance monitoring to enable profile-guided optimization (PGO) to effectively place heap-allocated objects in heterogeneous memory [80]. Though this framework is built on gcc, we suspect incorporating a similar approach into the TrackFM middle-end transformations would be straightforward.

4.5 Conclusion

We demonstrated that the compiler-based approach to far memory is a feasible path to automatically transform applications on software-based memory disaggregated systems. We realized the compiler-based approach with a prototype system called TrackFM, and demonstrated how it can outperform the kernel-based approach by up to 2× by merely recompiling the application. Its performance comes within 10% of the best performing library-based approach, AIFM, but requires no modifications to application code. TrackFM simultaneously achieves programmer transparency and good performance by leveraging novel compiler analysis and transformation techniques, and by using the highly-optimized AIFM runtime as a backend.

CHAPTER 5

CARDS: ENABLING POLICY DECISIONS DYNAMICALLY ON SOFTWARE DISAGGREGATED SYSTEMS

Chapter 4 demonstrates how compilers can automatically port applications to software memory disaggregated systems, thereby improving developer productivity. However, in TrackFM, many runtime policies, such as determining which objects should be remotable, were decided at compile time. Since the compiler lacks runtime information, TrackFM was forced to adopt conservative policies. For example, in TrackFM, all objects were assumed to be remotable. In this chapter, we present CARDS, a system that combines static and runtime information to determine far memory policies dynamically at runtime, without the need for profiling or making conservative decisions at compile time. CARDS stands for **C**ompiler **A**ssisted **R**emote **D**ata **S**tructures. Our approach builds on principles from library-based methods to determine far memory policies dynamically, with the key distinction that it is fully automatic and does not require any source code changes.

We summarize our contributions as follows:

- We demonstrate how compiler analysis can be combined with the runtime to automatically detect and transform data structures into remote data structures.
- We are the first to demonstrate how policy decisions can be made per data structure dynamically without the need for source code changes or profiling in far memory architectures.
- We demonstrate several policies to determine the selection of remotable data structures using information available to the compiler.
- We report on an extensive performance evaluation of CARDS using numerous micro benchmarks and real world applications.

5.1 CARDS Design

Existing legacy data structures can under perform in far memory architectures due to architectural mismatches, which differ substantially from single server systems [81]. A library-based approach like AIFM addresses this challenge by building custom remote data structures that have dedicated prefetching and remotable policies. However, the benefits of using custom remote data structures come at the cost of modifying an existing application. Prior work in far memory compilers like TrackFM and Mira addresses application modification by using static analysis and transformation, but the far memory policies in these systems are determined conservatively or necessitate the use of profiling. Our goal with CARDS is to enable profiling as a fallback when static analysis misjudges far memory policy decisions, allowing for more accurate, dynamic adjustments per data structure. Through its integration of compiler-identified data structures with runtime information, CARDS removes the need for manual application modifications or additional profiling efforts.

There are two fundamental challenges in using compilers to automatically transform existing application data structures into remote data structures. The first challenge is the loss of information that occurs when source code is compiled down to LLVM IR. The LLVM type system does not recognize user-defined data structure types, resulting in the loss of information about source code data structures. In particular, for a given load or store operation in LLVM IR, there is no direct way to determine which data structure the operation corresponds to. Identifying an application data structure enables the compiler to supply the runtime with both the data structure's access patterns and its memory allocations.

Our approach overcomes this limitation by building on prior work in inter-procedural data structure analysis (DSA) [44, 45]. Originally, DSA was designed to identify connectivity between memory objects across the entire program. Unlike shape analysis, which focuses on classifying data structures [44], DSA primarily concerns itself with how memory objects are connected within a program, thus improving compilation times for the analysis. In previous work, DSA has been used to capture properties of memory objects, such as determining whether they are type-safe or understanding how they relate to one another. In our work, we leverage DSA to automatically identify disjoint data structures, enabling their transformation into remotable data structures [45, 82]. Additionally, DSA is context-sensitive, meaning it can identify specific instances of a data structure. This capability is particularly useful for our purposes, as it allows us to assign unique prefetching and remotable policies to each instance.

To effectively enforce our policies for each data structure at runtime, we need a way to link the compiler-identified data structures to the application's allocation requests and memory access patterns. For example, during a memory allocation request, such as `malloc`, CARDS must determine which application data structure initiated the request to decide if it should be remotable. Fortunately, prior work on pool allocation [82] demonstrates how disjoint memory pools can be passed to the runtime by leveraging data structure analysis. Specifically, the pool allocation technique partitions heap data structure instances into pools and passes pool handles to the runtime, allowing for memory optimizations tailored to the data structure instances. In CARDS, we implement this pool allocation technique to establish far memory policies for each data structure.

The second challenge is that the allocation size of data structures are not always known at compile time. In CARDS, we develop policies that are co-designed with the runtime to address the lack of runtime information at compile time. For example, the policy for determining whether a memory allocation should be remotable is evaluated at runtime in CARDS. This is done by using compiler data structure information to identify the data structure responsible for the allocation, while also considering runtime details about the available local memory to decide if the allocation should be remotable.

Figure 5.1 describes the architecture of CARDS, where a developer is only re-

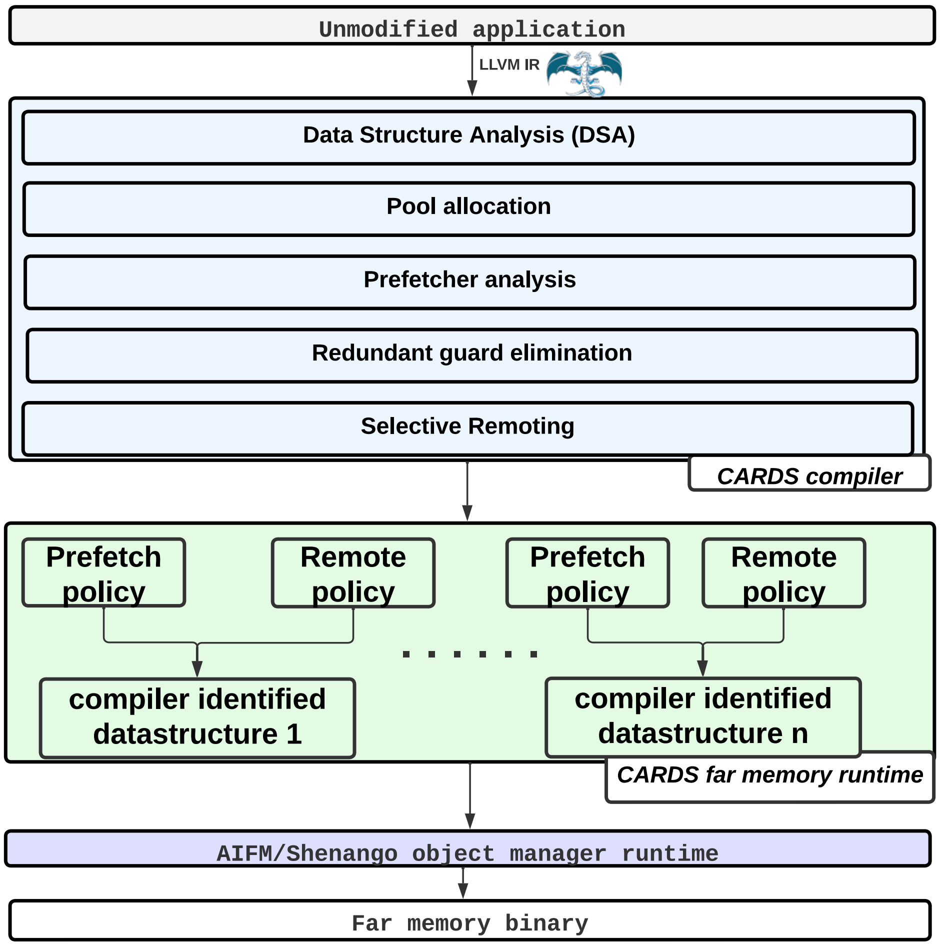


Figure 5.1. Users recompile their code with CARDS, resulting in an automatically transformed application that runs on a far memory cluster.

quired to compile their application with the CARDS compiler. We describe in detail on the CARDS compiler and runtime implementations in Section 5.2. Below, we outline some of the key design components of the CARDS system. The primary policies that affect data structures in far-memory architectures include the remotable policy, which determines whether a data structure's memory can reside on a remote server; the prefetching policy, which decides which objects to prefetch based on the data structure's access pattern; and the evacuation policy, which identifies which objects can be moved to the remote server for a data structure. In this paper, we focus on the first two: remotable and prefetching policies. The remaining policies will be integrated into the CARDS framework in the future.

Each data structure identified by the CARDS compiler is assigned its own prefetching and remotable policy. CARDS leverages static and runtime information available for a given data structure to determine whether a memory allocation request should be a candidate for remote memory. If a memory allocation is not a candidate for remote memory, then accessing a given memory address will not incur any network overheads and can eliminate memory safety checks (guards) , which are required to ensure safe access to an object.

A good selection of data structures will reduce network overheads and enable applications to run efficiently on far memory systems. Previous far memory compilers do not support this capability during runtime, instead they rely on profiling to make this choice or adopt a conservative approach (i.e., all objects are marked remotable). We will see later in Section 5.2 that having a conservative remotable policy where all objects are marked remotable, can incur high network and static instrumentation costs. The primary goal of CARDS is to demonstrate that we can determine efficient remotable policies for each data structure at compile time, without relying on runtime information about the size of a data structure, which previous far-memory compilers have failed to achieve.

CARDS applies different prefetching algorithms to individual data structures. By isolating these data structures during compilation, CARDS makes it easier to implement prefetching, since the algorithms only need to focus on the specific data structure without worrying about unrelated memory accesses. Additionally, CARDS allows switching between prefetching algorithms based on static or runtime information. The prefetching manager uses compiler data, such as detecting whether a data structure has a sequential access pattern, to choose the best prefetching strategy. This approach takes advantage of the compiler's ability to separate disjoint data structures, improving the effectiveness of the prefetching algorithms.

5.2 Implementation

We first outline the compiler passes in CARDS that automatically detect application data structures and transforms them into remote data structures. Following this, we describe the remotable and prefetching policies implemented in CARDS, which do not require application modifications or profiling.

Listing 5.1 contains two data structures (`ds1`, `ds2`) that are being initialized, with `ds2` specifically being initialized within a loop. We use this example to demonstrate how the CARDS compiler and runtime information are combined to automatically transform data structures and make appropriate policy decisions dynamically at runtime, as discussed in the following sections.

5.2.1 Compiler optimizations.

CARDS utilizes NOELLE to build its compiler passes [58]. NOELLE offers high-level, program-wide abstractions built on top of LLVM IR [59], which significantly reduces the time needed to develop compiler analyses and transformations in LLVM.

DSA Pass CARDS leverages SeaDSA [45] in order to automatically detect disjoint data structures within an application. The data structure analysis in SeaDSA is inter-procedural and context-sensitive, enabling it to capture more data structures than the original DSA [82].

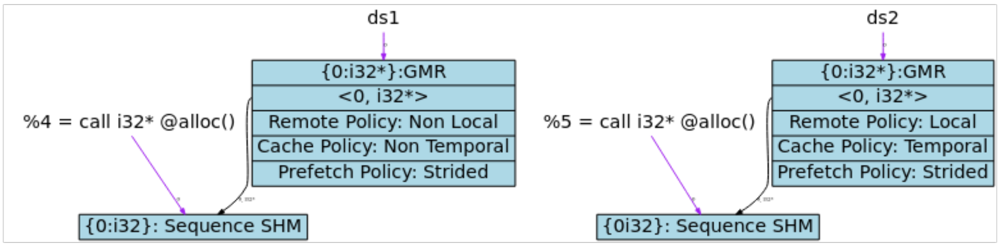


Figure 5.2. DSA for Listing 5.1

Figure 5.2 shows the disjoint data structures identified for Listing 5.1 by the CARDS compiler. Notably, only heap-allocated data structures are identified by our analysis.

Listing 5.1. Example C code with 2 data structures initialized

```
int * ds1, * ds2;
double * alloc() {
  return malloc(ARRAY_SIZE);
}
void main() {
  ds1 = alloc();
  ds2 = alloc();
  Set(ds1, 0);
  Set(ds2, 1);
  for (int k=0; k<NTIMES; k++)
    Set(ds2, k);
}
void Set(int * ds, int val) {
  for (j=0; j<ARRAY_SIZE; j++)
        ds[j] = val;
}
```

Pool Allocation After identifying disjoint data structures, then CARDS transmits this data structure information to the runtime by utilizing the pool allocation algorithm 1. We re-implement pool allocation within the NOELLE framework, because the original implementation of the pool allocation algorithm [82] had not been maintained for newer compiler versions. Unlike the original Pool Allocation Algorithm, which employs bottom-up interprocedural analysis to identify disjoint data structures, CARDS uses context-sensitive disjoint data structures identified by DSA and passes them to the pool allocation algorithm 1. The Pool Allocation algorithm 1 is used to initialize data structures and link memory alloca-

Algorithm 1 Pool Allocation Algorithm [82]

```
 1: for all $F \in$ functions($P$) do
 2:     seadsa::graph G = DSAGraphForFunction(F)
 3:     for all $n \in$ Nodes($G$) do
 4:         if ($n.alloc == Heap\ and\ escapes(n)$) then
 5:             $ds\_handle \leftarrow AddDSHandleArg(F,n)$
 6:             $dsmap(n) \leftarrow ds\_handle$
 7:             $argnodes(F) \leftarrow$ argnodes(F) $U$ $\{n\}$
 8:         else// Node is local to fn
 9:             $ds\_handle \leftarrow DS\_INIT(F,n)$
10:             $dsmap(n) \leftarrow ds\_handle$
11:         end if
12:     end for
13: end for
14: for all $F \in$ functions($P$) do
15:     for all $I \in$ instructions($F$) do
16:         if $I\ isa\ malloc\ callinst$ then
17:             replace I with dsalloc(size,dsmap(N(ptr)))
18:         else if $I\ isa\ callinst$ then
19:             for all $n \in$ ArgNodes($Callee$) do
20:                 $addCallArg(dsmap(NodeInCaller(F,I,n)))$
21:             end for
22:         end if
23:     end for
24: end for
```

tions to their corresponding data structures. It works in two phases. In the first phase (lines 1–13), the algorithm modifies functions in the program that might allocate memory on the heap. This includes both direct calls to `malloc` and indirect calls that eventually lead to `malloc` The algorithm uses a map, `dsmap`, to track a handle for each data structure. If a function returns a pointer or passes it outside the function (escaping pointer), the algorithm adds extra arguments to the function to handle the data structure properly. If the pointer does not escape, the algorithm calls the CARDS runtime to initialize the data structure, and the handle is saved in `dsmap`. In the second phase (lines 14–24), the algorithm updates the functions to include the data structure handles from `dsmap` as arguments. This ensures that any function that calls the modified functions from the first phase knows which data structure it is working with, so memory can be managed correctly at runtime. For more details on pool allocation, see the Pool Allocation paper [82].

Listing 5.2. CARDS pool allocation compiler transformation for Listing 5.1

```
...
double * alloc(int DH) {
  return cards_malloc(ARRAY_SIZE, DH);
}
void main() {
...
   //ds_init(ds_id, cache_policy, prefetch_policy,
  remote_policy)
  int dh1 = ds_init(1, false, STRIDED, REMOTABLE);
  int dh2 = ds_init(2, true, STRIDED, LOCALIZE);
  ds1 = alloc(dh1);
  ds2 = alloc(dh2);
...
}
...
```

Additionally, similar to the Pool allocation algorithm, we manage indirect pointers by utilizing equivalence classes, as described in [82]. Each disjoint data structure identified by the compiler receives a unique identifier known as the data structure handle, which the runtime uses to differentiate between various data structures and dynamically apply policies for each one. At runtime, each identified data structure has its own prefetching manager and specific remotable policy. Additionally, when a data structure is created, we pass supplementary information, such as its “hotness” based on usage in functions, by employing compiler call graph analysis on strongly connected components. This approach enables the runtime to utilize compile-time information when determining the remotable and prefetch policy for each data structure.

Listing 5.2 illustrates how CARDS pool allocation transforms Listing 5.1 and employs pool allocation to segregate data structures with different prefetch and remotable policies. Each data structure is assigned a unique handle, `DH`, that is appended to the non-canonical bits of a pointer, facilitating the mapping of addresses to their corresponding data structures.

Prefetching analysis In this pass, we gather application type information within LLVM IR and utilize induction variable analysis to identify sequential access patterns. The CARDS compiler analysis operates at the level of individual data structures, with each data structure assigned its own prefetch policy. Figure 5.2 highlights the data structures with strided access patterns identified at compile time for Listing 5.1.

Redundant Guard Elimination In a far memory system, memory accesses to data structure objects can take place either when the objects are located in local memory or in remote memory. In line with previous far memory compiler systems, CARDS automatically inserts guard checks on memory instructions to ensure memory safety. Like TrackFM [42], CARDS employs static analysis to eliminate unnecessary guards. However, TrackFM compiler optimizations rely on induction variables to eliminate guards which do not work for recursive data structures such as linked lists and hash maps. In contrast, CARDS can elim-

Listing 5.3. CARDS redundant guard elimination transformation for Listing 5.1

```
void Set(int * ds, int val) {
  for (j=0; j<ARRAY_SIZE; j++) {
    if ((&ds[j] < ds_lb) || (&ds[j] > ds_ub)) {
       ds_addr = cards_deref_addr(&ds[j])
       update_bnds(&ds[j], ds_lb, ds_ub);
    }
  *ds_addr = val;
  }
}
```

inate redundant guards even for recursive data structures by employing temporary stack variables for each data structure. These variables monitor pointer accesses and record the lower and upper bounds of objects that have already been localized. When accessing an object within a given data structure, CARDS first checks if this object has been resolved previously. If it has, the resolved object address is used directly; otherwise, the runtime is invoked. We only inject a call into the runtime when the bounds for an object are deemed invalid.

Listing 5.3 presents the pseudocode for CARDS's loop transformation applied to Listing 5.1. Additionally, we have found that the O3 compiler passes, such as mem2reg, can often promote these temporary stack variables (`ds_lb`, `ds_ub`) to registers for many applications, further reducing the overhead associated with guard checks.

Selective remoting One of the challenges in selectively marking data structures as remotable (described in Section 5.1) is that the size of a dynamic data structure is not always known at compile time or during runtime memory allocation (resulting in partial information). This knowledge is crucial to prevent the localization of data structures whose memory sizes exceed the capacity of a single server. Listing 5.4 demonstrates our transformation applied to Listing 5.1. Before executing a loop, the CARDS compiler injects a call into the runtime to check whether the data structures used within the loop are remotable. If all the data structures are marked as non-remotable, the execution branches to the uninstrumented version.

Listing 5.4. CARDS selective remoting compiler transformation for Listing 5.1

```
void Set(int * ds, int val) {
  remote = check_remotable_policy(ds);  //loop v1
  if (remote) {
    for (j=0; j<ARRAY_SIZE; j++) {
      if (safe_to_access(&ds[j]))
        ds[j] = val;
    }
  }
  else {
    for (j=0; j<ARRAY\_SIZE; j++) //loop v2
      ds[j] = val;
  }
}
```

The advantage of treating data structures as non-remotable is that they can execute locally without any instrumentation. However, this approach is not always feasible at compile time due to a lack of runtime information. Since the memory layout of the data structure is not known at compile time, it is impossible to determine whether the data structure should be marked as remotable during compilation. This limitation can be mitigated through profiling or using techniques like loop peeling, where certain iterations of a loop are peeled out. These peeled iterations can be used at runtime to help decide whether the larger loop should be remotable. However, determining the number of profiling runs or loop iterations needed to make an accurate decision is not always straightforward. Instead, in CARDS, we use code versioning by maintaining two versions of the code: one that is instrumented and

Table 5.1. Comparison of primitive overheads for CARDS and TrackFM. Costs are reported in median cycles over 100 trials.

Runtime Event	**Local Cost**	**Remote Cost**
CARDS read fault	378	59K
CARDS write fault	384	59K
TrackFM read guard	462	46K
TrackFM write guard	579	47K

another that is not. The uninstrumented version is selected only if all data structures are localized, ensuring that unnecessary instrumentation is avoided when not needed.

5.2.2 Runtime.

CARDS utilizes a modified version of the AIFM runtime to manage far memory objects at the granularity of data structures. During compile time, CARDS identifies disjoint data structures and assigns each a unique data structure ID, which is employed at runtime, as illustrated in Listing 5.2. The data structure ID is appended to the non-canonical bits of pointer addresses during memory allocation calls to manage these associations. CARDS monitors cache hits and misses for each memory object, leveraging these statistics on a per-data structure basis to inform runtime policy decisions. Unlike previous compiler-based far memory systems, CARDS allows policy decisions such as prefetching and remotability at the individual data structure level, offering finer control over memory management.

Additionally, CARDS employs a customized version of the standard libc library to link memory addresses with specific data structures. Listing 5.2 demonstrates how CARDS transforms memory allocations and incorporates `ds_init` calls. The `ds_init` method is injected before any memory object associated with a given data structure is accessed, utilizing reachability analysis within the CARDS compiler pass. These data structures are initialized in the order they appear in the program by invoking the `ds_init` method, which configures each data structure according to the policies informed by the compiler.

```
0: shr    $0x30,%rcx          // custody check (is this CARDS-managed ptr?)
1: je     4                   // if not, perform original load/store
2: callq  <card_deref_fn>     // otherwise, runtime call
4: mov    %rbx,(%rax)         // TARGET LOAD/STORE
```

Figure 5.3. CARDS guard lowered to x64 code

CARDS guards CARDS manages far memory at the object level, allowing for objects of arbitrary sizes to reside in either local or remote memory. To ensure memory safety, it is essential to localize an object before access. This safety is achieved in CARDS by injecting guards on memory accesses to data structure objects.

In CARDS, an object may map to multiple addresses, determined by its size. The size of an object is guided by compiler hints provided to the runtime during the data structure initialization process (`ds_init`). For instance, a declaration like `char ds[4096]` could correspond to a single CARDS object if the object size for the data structure is set to 4K. Consequently, CARDS data structures can have varying object sizes based on the static hints given by the compiler.

Figure 5.3 illustrates a CARDS guard check. If a memory address has its non-canonical bits (bits 48-63) set (known as a custody check), CARDS injects a call to the `cards_deref function`. The `cards_deref` function is responsible for ensuring memory safety within CARDS. This function first maps the higher-order address bits to their corresponding data structure and then uses the lower address bits to associate with the actual object. Following this mapping phase, the system checks whether an object is localized; if it is not, CARDS calls into the AIFM runtime to fetch the object.

The CARDS compiler injects guard checks once for every object access within a loop. If multiple memory locations map to the same object, a check occurs only once, thanks to the redundant guard optimization described above.

Listing 5.5. CARDS deref function

```
uint64_t cards_deref(uint64_t addr) {
        //get ds handle from non canonical bits
        uint64_t ds_id =  (addr >> ORT_POS);

        DS * ds = ds_list[ds_id];
        //map address to object
        uint64_t ind = off >> ds->obj_shift;
        FarMemPtr * obj = ds->pool_manager->ptrs_[ind];

        //if object already in local memory
        if (safe_to_access(obj))
                return obj.paddr;

        //fetches object over the network
        LocalizeObject(ds, ind, obj);
        return obj.paddr;
}
```

Remotable policy selection CARDS manages local memory by dividing it into two categories: pinned memory, which cannot be remoted, and remotable memory, from which data structures marked as remotable are allocated. The system uses a custom libc library to control memory allocations, associating each allocation with a specific data structure. During a memory allocation, CARDS evaluates static information about the data structure, provided by the compiler, to determine if the memory should be allocated from remotable memory.

The main challenge in deciding whether a data structure should be non-remotable is that its "hotness" (locality) and size may not be known at compile time. This uncertainty makes it difficult to determine which data structures should be localized. Previous compiler approaches have addressed this by using profiling to determine the hotness and size of memory objects, and then, based on this information, deciding whether these objects should be placed in remotable memory. For instance, Mira [43] uses a memory profiler to determine allocation sizes, and only objects with large sizes are further analyzed to decide their memory policy.

However, without profiling, determining the hotness of data structures is difficult. Since the locality of a data structure often depends on the program's input, it is hard to predict at compile time. In CARDS, we attempt to approximate this "hotness" using several policies. We consider a data structure hot if it has a long lifetime or is accessed frequently in memory operations. These policies do not rely on data structure size and use a tunable threshold, `k`, to decide what percentage of data structures should be allocated to non-remotable memory. Ideally, `k` is set higher when more local memory is available and lower when memory is limited, making the approach more dynamic and adaptable.

Below, we outline several policies that can improve the selection of data structures to be localized based on static information:

Linear Assignment This policy allocates pinned (non-remotable) memory sequentially in program order. When local memory is exhausted, it switches to using remotable memory.

Random Assignment This policy allocates pinned (non-remotable) memory randomly across memory allocations.

Maximum Reach of Data Structures This policy targets data structures used in functions with longer lifetimes, marking them as local (non-remotable) during compilation. We estimate lifetime by identifying data structures accessed by the top `k` functions with extensive caller/callee chains. CARDS uses the strongly connected components (SCC) call graph to track which functions access a data structure. If pinned memory becomes full, CARDS will fall back to using remotable memory.

Maximum Uses of Data Structures

$$ds = MAX(\#loops + \#functions)$$

This policy focuses on data structures that are accessed most frequently in memory operations, such as loads and stores, within loops and functions in LLVM IR. These data structures are marked as non-remotable candidates. The top `k` most frequently accessed data structures are then designated as non-remotable to ensure that frequently used data remains local.

Due to limited runtime information, it is sometimes impractical to make accurate decisions about remoting at compile time. To address this, the CARDS runtime can override static hints as needed. For example, if a data structure does not fit in local memory despite a static hint for localization, the runtime may choose to remote it. In cases where dynamic data structures gradually grow during execution, the runtime tracks allocations to ensure they remain local. This allows certain code paths to execute without instrumentation for non-remotable data structures. When a data structure is remoted, CARDS automatically switches to an instrumented code path.

In Listing 5.1, we observe that `ds2` has a higher usage than `ds1` and benefits more from localization. By implementing the policy that prioritizes data structure usage, CARDS can make more informed decisions; using the linear policy may lead to suboptimal choices. In Figure 5.4, we compare various policies for Listing 5.1, allowing one of the data structures to be localized based on compiler policy by setting `k` = 50%. Both data structures in Listing 5.1 are allocated 3GB of memory. When 50% of local memory is available, one of the data structures can be localized. A naive policy would localize `ds1`, while a refined policy correctly localizes `ds2`, resulting in improved performance by minimizing network communication, as shown in Figure 5.4.

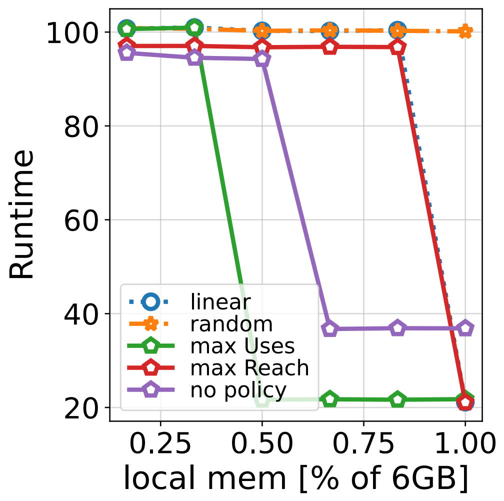


Figure 5.4. CARDS performance across different remotable policies for Listing 5.1 when top k=50% of data structures are marked as non-remotable.

Prefetching Policy Selection A variety of prefetching policies are available, each differing in complexity, specificity, and aggressiveness. For CARDS, we have chosen to support existing compiler prefetchers, including a majority stride-based prefetcher, a greedy recursive prefetcher, and a jump pointer prefetcher [83]. Based on the static and dynamic information available for each data structure, CARDS selects the most appropriate prefetch policy. Standard prefetching metrics, such as accuracy and coverage, are used to evaluate the effectiveness of each prefetching policy. The combination of static and dynamic information per data structure creates exciting research opportunities for advancing prefetching algorithms within CARDS.

5.3 Evaluation

CARDS combines static and runtime information to determine optimal far-memory policies for each application data structure. Our evaluation assesses the performance impact and advantages of combining compiler and runtime optimizations at the granularity of data structures, enabling informed decision on remotable and prefetching policies, using micro benchmarks. We then demonstrate that CARDS serves as a viable, non-profiling alternative, outperforming conservative methods while maintaining acceptable performance relative to profiling-based approaches. Our evaluation aims to address the following questions:

- How do CARDS remotable policies benefit applications? (§5.3.1)
- How do CARDS prefetching policies benefit applications? (§5.3.2)
- How does CARDS perform on realistic applications? (§5.3.3)

Experimental setup We conducted our experiments on CloudLab [64] using two x170 machines with 10-core Intel Xeon E5-2640v4 CPUs clocked at 2.40 GHz, 64GB RAM and a 25 Gb/s Mellanox ConnectX-4 NIC. We used Ubuntu 18.04 with Linux kernel version 5.0 and DPDK version 18.11 (used by AIFM). CARDS builds on LLVM version 14.0.6, [30] with NOELLE v14.1.0.[31] For large codebases we use WLLVM[32] to produce bitcode for the entire application before passing it to the CARDS compiler.

Applications We select three benchmarks to evaluate CARDS, among which the analytics benchmark and BFS represent common access patterns in datacenters.

NYC analytics is a data analytics application that uses the 2014 NYC taxi trip dataset from Kaggle [33] to analyze New York City taxi trips. We choose this benchmark to compare against existing far-memory compilers (both profiling and conservative) and validate our results against TrackFM [42]. For the profiling the Mira compiler, we were

[30]commit f28c006

[31]commit 68f334a

[32]`https://github.com/travitch/whole-program-llvm`

[33]`https://www.kaggle.com/code/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook`

unable to reproduce the NYC benchmark due to the incomplete Mira implementation[34]; instead, we use a projected curve based on the results from their paper [43]. The memory working set size for the taxi-trip workload was 31GB, and the dataset size was 16GB.

Next, we choose a widely used benchmark suite, PolyBench, which is a collection of benchmarks with static control parts [84]. Within PolyBench, we select *ftfdapml* (Finite Difference Time Domain Kernel using Anisotropic Perfectly Matched Layer), which is used to simulate optical devices and model electromagnetic wave interactions with materials. We select ftdapml because it has the largest number of data structures in the PolyBench suite, making it useful for evaluating remoting policies in CARDS. The memory working set size of ftdapml is 8GB.

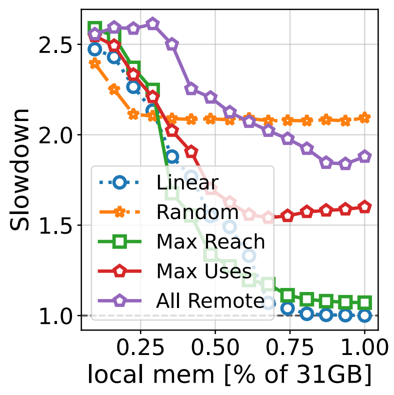
Finally, we select *BFS*, a graph processing workload commonly used in datacenters. BFS is characterized by an irregular access pattern, which is typical of datacenter workloads. We use the BFS benchmark from the GAPS benchmark suite [85]. The memory working set size for BFS is 1.2 GB.

5.3.1 Evaluation of CARDS remoting policy.

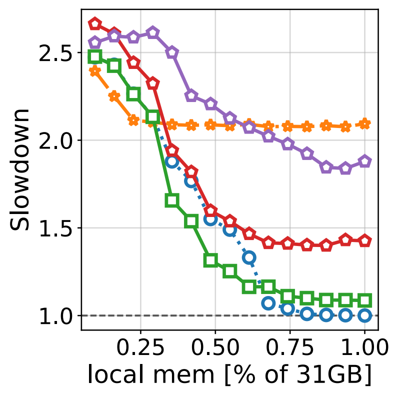
In far memory systems, ideally, hot data structures should not be remotable (i.e., they should use pinned local memory) to avoid the cost of network fetches and to mitigate guard costs. However, marking a data structure remotable becomes necessary if their memory requirements exceed a single server's capacity. In this section, we evaluate CARDS's remoting policies, which are determined dynamically without profiling or conservative assumptions, and compare them with existing profiling and conservative compilers.

Figures 5.5 to 5.7 compares the remoting policies of CARDS (as discussed in Section 5.2) across three application benchmarks. As local memory increases, more data structures are designated as non-remotable, meaning their memory allocations are pinned to

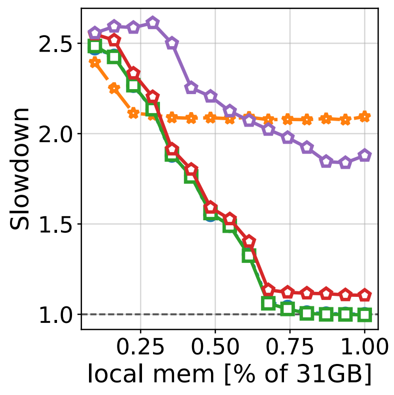
[34]`https://github.com/WukLab/Mira`



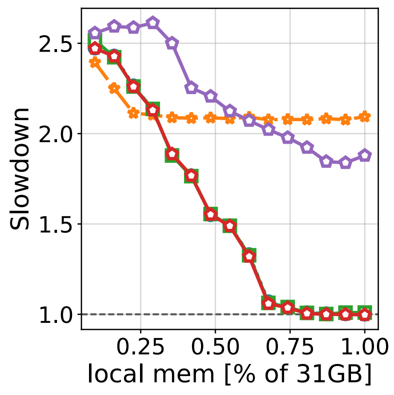
(a) top k = 25%



(b) top k = 50%



(c) top k = 75%



(d) top k = 100%

Figure 5.5. CARDS remoting policies for analytics benchmark where we increase the amount of data structures to be localized from left to right, notice that max reach policy is unaffected even when data structures are localized only in top 25% of functions.

local (non-remotable) memory. These figures show that selectively remoting data structures can improve application performance by up to $\sim 2\times$, especially when sufficient local memory is available.

For the analytics workload, CARDS allocates available local memory by setting aside 1 GB for remotable memory, while the remaining local memory is used for pinned memory, if available. CARDS identifies `22` disjoint data structures at compile time, each of which is assigned a dedicated prefetcher and a custom remote policy manager. Since the sizes of these data structures are not known at compile time, CARDS includes a tunable parameter, `k`, that controls the percentage of data structures to be localized.

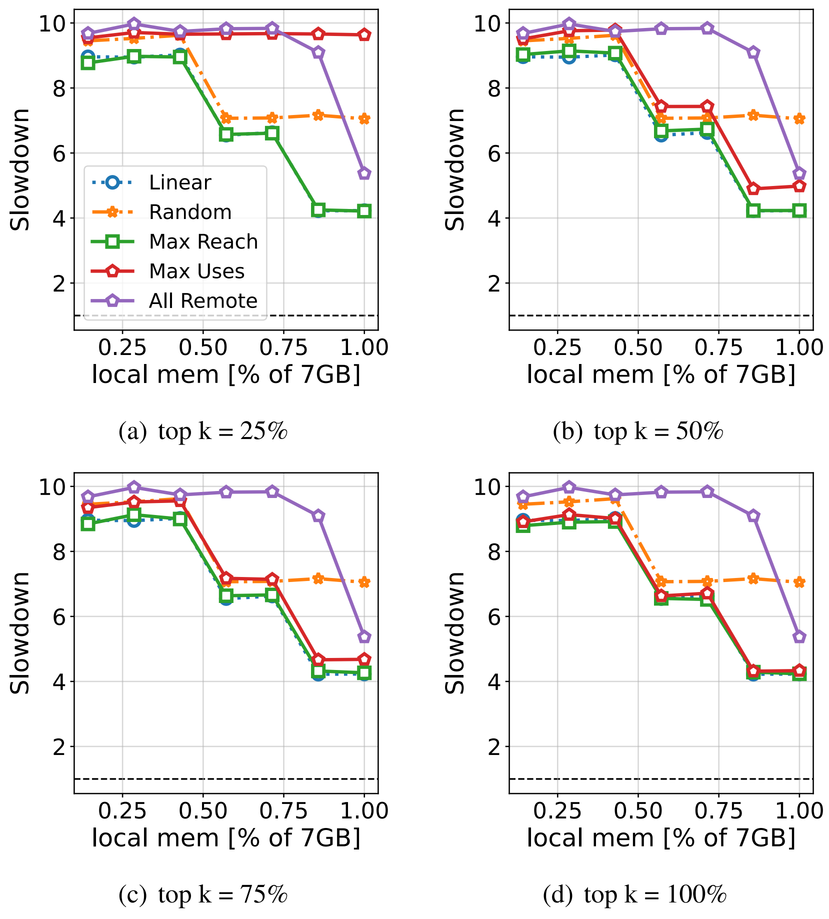
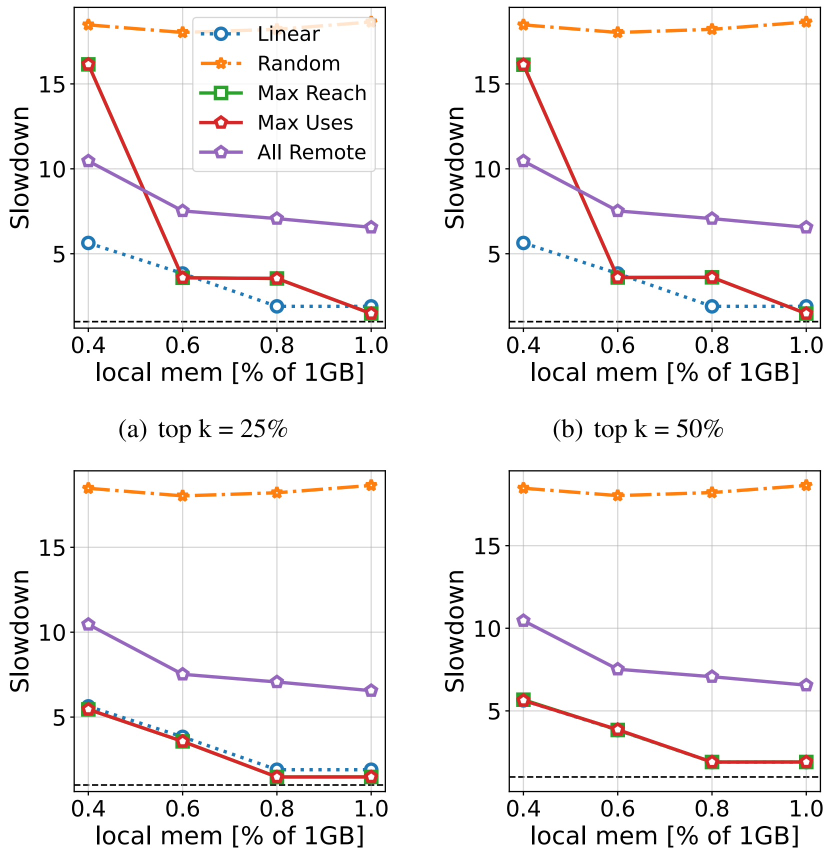


Figure 5.6. CARDS remoting policies for ftfdapml where we increase the amount of data structures to be localized from left to right, notice that max reach policy is unaffected even when data structures are localized only in the top 25% of functions.

When all data structures are localized (k = 100), the policies exhibit similar performance, except for the random policy. However, as the number of localized data structures decreases, the effectiveness of each policy varies. The max reach policy proves to be more resilient to selective remoting. In scenarios where the application's local memory exceeds 90%, a linear policy suffices because all data structures can be localized on demand. Unlike other policies that statically assign certain data structures to be remotable, the linear policy makes decisions at runtime.

In Figure 5.6 we allocate 1 GB of remotable memory for the ftfdapml benchmark, with the remainder designated for pinned (non-remotable) memory. CARDS identifies 15



(a) top k = 25%

(b) top k = 50%

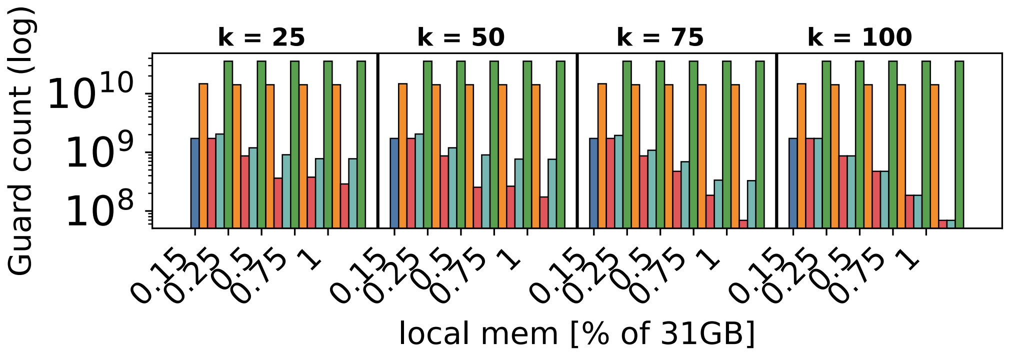
(c) top k = 75%

(d) top k = 100%

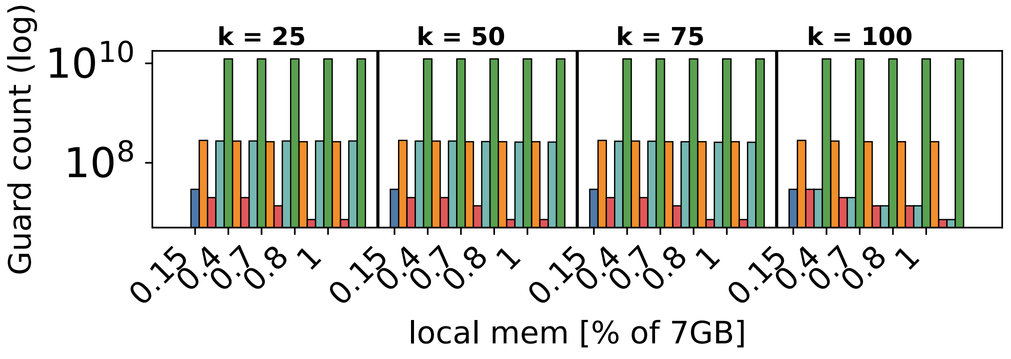
Figure 5.7. CARDS remoting policies for BFS where we increase the amount of data structures to be localized from left to right, notice that the linear policy is unaffected even when the top 25% data structures are localized.

disjoint data structures at compile time, with certain data structures requiring more memory than others. The max uses policy performs better when `k` exceeds 25%. Additionally, both the linear and max reach policies demonstrate greater tolerance to selection changes, achieving performance that is approximately ~4× better than the all-remotable configuration.

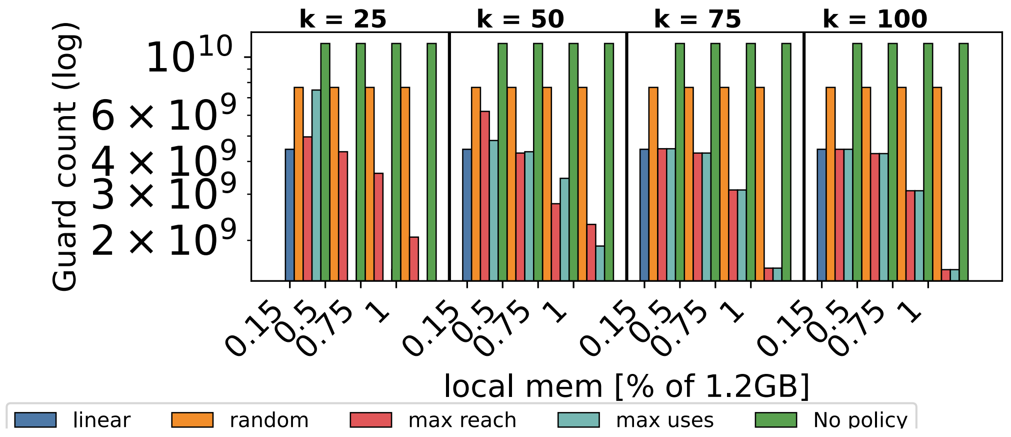
In Figure 5.7, the BFS benchmark is allocated 256 MB for remotable memory, with the rest designated for pinned memory. CARDS identifies `19` disjoint data structures, with varying memory requirements among them. The linear policy consistently outperforms other policies across different selections of data structures.



(a) NYC analytics



(b) ftfdapml



(c) BFS

Figure 5.8. CARDS eliminates a significant number of guards due to its remoting policies determined at runtime across benchmarks.

Figure 5.8 illustrates that selectively marking data structures as remotable can significantly reduce network and guard overheads. When all data structures are marked as remotable, approximately 10 billion guard checks are performed across the three benchmarks, which can become prohibitively expensive when sufficient local memory is available. Importantly, these policies not only eliminate guard checks but also reduce network communication for these data structures.

In summary, Figures 5.5 to 5.8 demonstrate that, with the exception of the random policy, all other policies enhance the efficiency of far memory systems compared to the conservative approach of marking all data structures as remotable candidates in far memory compilers. When objects are conservatively designated as remotable at compile time, they can lead to performance overheads in both scenarios: when local memory is limited and when it is adequate. Specifically, if local memory is constrained and all data structures are marked as remotable, the system faces frequent network requests to retrieve objects. On the other hand, when local memory is plentiful, the system suffers from the overhead associated with static guard checks. In both situations, marking all objects as remotable can result in considerable network and instrumentation overhead in far memory compilers. Therefore, selectively remoting data structures using even simple policies like linear assignment proves to be more effective than a conservative approach of marking all data structures as remotable.

5.3.2 Evaluation of CARDS prefetch policy.

CARDS identifies disjoint data structures within an application, with each structure having its own dedicated prefetcher. We assess the advantages of this separation by comparing CARDS to the TrackFM compiler. In Figure 5.9, we analyze various data structures that perform the sum operation (`c[i] = a[i] + b[i]`). The memory working set size is set to 7 GB, and we measure the speedup of CARDS relative to TrackFM. Our findings show that data structures with induction variables, such as array sums, run efficiently om

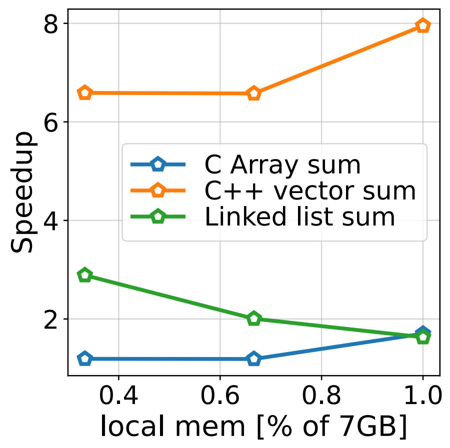


Figure 5.9. CARDS speedup compared to using TrackFM for pointer chasing data structures, as CARDS can identify disjoint data structures which have their own prefetchers, CARDS outperforms TrackFM consistently.

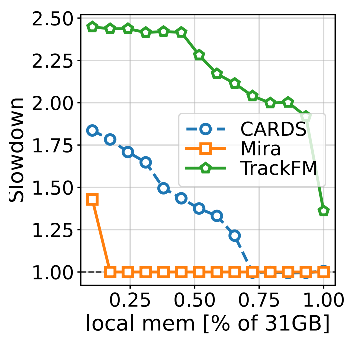
TrackFM. However, for pointer-chasing benchmarks like C++ vectors and maps, CARDS consistently outperforms TrackFM. This is due to the fact that TrackFM relies on induction variables for prefetching, which are not available in pointer-chasing applications.

5.3.3 Benefits of CARDS on real world applications.

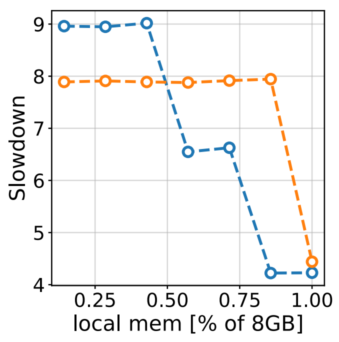
Figure 5.10 compares the performance of CARDS with prior far memory compilers. We observe that CARDS achieves performance that falls between profiling and non-profiling compilers for the analytics benchmark. Notably, CARDS consistently outperforms TrackFM by up to approximately 2×, especially when more than 50% of the local memory is available. With 25% of non-remotable memory, CARDS is within 20% of the performance of Mira (profile-based compiler). However, as the available local memory increases, Mira surpasses CARDS. We aim to explore improved policies to close this gap further in future work. For the other benchmarks, CARDS consistently outperforms TrackFM.

5.4 Discussion and Future Work

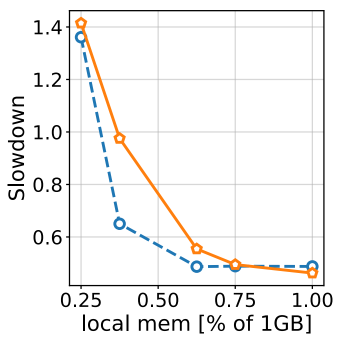
We aim to convey the lessons learned, potential future work, and limitations en-



(a) NYC



(b) ftfdapml



(c) BFS

Figure 5.10. CARDS performance compared to prior far memory compilers, CARDS is within 20% of Mira when local memory is less than 25%, and outperforms TrackFM consistently when more than a few data structures are localized.

countered while developing CARDS.

5.4.1 Lessons.

During our investigation into the selective assignment of remotable data structures, we observed that our compiler transformations could interfere with existing compiler optimization passes, leading to performance degradation, particularly for data structures that were not instrumented. This observation prompted us to explore loop versioning. By maintaining both instrumented and uninstrumented versions of the code, existing compiler optimizations could enhance the performance of the uninstrumented code. This approach enabled CARDS to achieve near native performance when sufficient local memory was available.

5.4.2 Limitations.

External Library Support Currently, CARDS lacks support for external libraries and necessitates that the entire application, including external library code, be compiled with CARDS. However, we believe this limitation can be addressed in several ways. One potential solution is to manage data structures from external libraries through the kernel without instrumentation by CARDS. This can be implemented on systems like Atlas [86], which allow for the coexistence of object-based and kernel-based runtimes, enabling the kernel to manage data structures from external libraries. Alternatively, similar to SeaDSA [45], we could provide stub functions for external functions that describe how they modify memory, facilitating their transformation by CARDS.

Function Versioning While CARDS benefits from loop versioning and avoids the overheads of static instrumentation, certain code paths may not utilize loops. We envision that function versioning could address this issue, although it requires careful consideration regarding which functions should be versioned to prevent excessive code size expansion.

5.4.3 Additional Policies for Far Memory.

Although we have explored only two policies at the data structure granularity, we believe the following strategies could also be valuable for far memory systems:

Evacuation Policy CARDS can utilize compiler analysis to approximate the lifetimes of data structures, allowing for the asynchronous evacuation of dead structures and improving application efficiency. Techniques from context-sensitive, liveness-based garbage collectors [87] can be applied to the data structures managed by CARDS.

Placement of Data Structures Across Memory Pools CARDS can optimize the placement of data structures in various memory pools based on the characteristics of code and data access patterns that best align with the memory device capabilities. This strategy enhances locality and prefetching algorithms by grouping closely related datasets within the same memory pool.

Choice of Network Interconnect Per Data Structure Each data structure in CARDS could utilize different network transport protocols, such as UC, RC, or UDP, tailored to the characteristics of the data structure. Mira [43] has examined multiple transports at the cache granularity; it would be interesting to investigate how data structures could benefit from one-sided or two-sided communication in a far memory context.

Data Structure Management by Kernel or Compiler Certain data structures benefit from temporal locality and are best managed by the kernel. In CARDS, it is feasible for the compiler and the kernel to manage separate data structures, enhancing overall application performance. Furthermore, individual objects within a data structure could be managed by either the kernel or the compiler concurrently. Implementing this would require running CARDS on Atlas [86], which supports such hybrid designs.

5.5 Conclusion

We present CARDS, a prototype compiler-assisted far memory system that automatically transforms data structures into remote data structures. Through experiments, we have demonstrated that efficient remotable and prefetching policies can be determined dynamically, without profiling, by integrating runtime and static information at the data structure level. CARDS outperforms previous non-profiling compiler systems by up to ~2× and comes within 25% of the performance of profiling-based compilers when local memory is constrained.

CHAPTER 6
RELATED WORK

I begin by describing related work on OS level disaggregation within single server in Section 6.1, followed by related work on memory disaggregation in Section 6.2 and finally describe related work on data structures for far memory in Section 6.3.

6.1 OS-level disaggregation

As far as we are aware, we are the first to model speedup analytically for multi-OS environments. We refer readers to Gioiosa et al. for an excellent empirical study of system call delegation [31]. Our work was inspired by Amdahl's original formulation of parallel speedup [33], Gustafson's refinement [34], and Sun and Ni's extended model for incorporating memory-bound programs [35]. One might view a multi-OS setup as a general distributed system, where forwarded system calls are simply treated as RPCs. In its simplest case, such a system might suitably be modeled using a LogP model [88]. However, models like LogP primarily relate to the communication/computation ratio, and furthermore do not consider the asymmetry between execution times and system interfaces in different operating systems. Our model, in contrast, is designed to capture this asymmetry. As we extend our speedup model to include concurrently executing SOS and GPOS threads, we intend to draw on existing models of parallel computation. Applications *limited* by system call usage can be viewed through a lens of "operational intensity," which others have visualized using roofline models [89, 90]. While roofline models are based on architectural characteristics rather than properties inherent to the application and system software stack, we believe they could be adapted to provide insight for multi-OS systems, and we plan to explore this further in follow-up work.

6.2 Memory disaggregation

Prior work on far memory primarily falls along two lines: software and hardware-based. Hardware-based approaches center on the idea of removing the limitation of the architected page size [71, 1, 91]. On commodity machines, however, such specialized hardware is not yet an option. Prior work on improving *software-based*, programmer-transparent, far memory focuses on overcoming the limitations of the kernel-based approach, either by using better prefetching strategies [92, 93], by reducing page fault costs in the kernel [7], or by using high-performance networking [8]. Significant benefits are available when full programmer transparency is not a requirement, as shown by AIFM [1] and Carbink, which focuses on fault-tolerant far memory [94].

One way to improve on the kernel-based approach is to leverage a custom OS. DiLOS focuses on mitigating software overheads (especially of the paging subsystem) by building a LibOS specialized for disaggregated memory [13, 40]. DiLOS, which builds on OSv [95], uses a custom, *unified page table* that incorporates remote page table entries in lieu of repurposing the traditional swap cache to track remote page state, thus reducing software overheads. This approach can actually outperform AIFM with sufficient prefetching, demonstrating that in some cases reducing the page fault costs can counteract the negative effects of I/O amplification. However, even though DiLOS can run unmodified binaries (through POSIX compatibility), adopting a new OS can be a challenge. TrackFM, in contrast, runs on stock Linux without any changes.

Meta's production-scale far memory framework (TMO) leverages run-time information to transparently offload memory onto heterogeneous storage, and demonstrates that far memory pays off at scale [12]. Far memory systems share lineage with a large body of work on distributed shared memory (DSM), as these systems are similarly constrained by the architected page size. Thus, there is also work in this domain on avoiding page fault overheads. For example, Blizzard [96] and Shasta [9] work at sub-page granularity to mitigate false sharing. User-space approaches to DSM that leverage the compiler employ optimizations such as aggregation/hoisting of guards to reduce overheads [60, 61, 62].

However, these systems assume that an entire allocation is localized at once. In our system, chunks of a large allocation can be in independent states (local or remote), making hoisting optimizations more challenging. Prior approaches also assume that localized memory will not be evacuated again, which we must handle. Many of the optimizations applied in DSM systems relate to synchronization overheads and communication avoidance [97, 98, 99, 100], which are not applicable to non-coherent, far memory setups. TrackFM requires more careful analysis to reducing guard overheads since the same assumptions made for user-space DSM systems do not apply. While unrelated to far memory, we build on ideas from prior work on using the compiler to replace paging-based address translation, namely CARAT [101] and CARAT CAKE [102].

6.3 Data structures for far memory

Hardware based approaches such as the work from Aguilera et al. explore data structures for far memory and introduce new hardware primitives to improve its efficiency [81]. Recently, the proposed CXL [103] interconnect provides hardware-based disaggregation at cache granularity. Similarly, CLIO [91] and KONA [71] introduce new virtual memory systems, custom network stack, offloading capabilities, and fine-grained page sizes in the hardware. Similarly, MIND demonstrates how memory management logic can be placed in network fabric to support shared memory coherence with performance [104]. Although, these approaches are promising, they are yet to be available for commercial use. Furthermore, we envision CARDS can be used as an emulator to guide intuition in such systems.

The compiler community has explored logical data structure analysis for SMP-based hardware. For example, pool allocation for disjoint data structures [82] and SeaDSA [45] have shown the benefits of compiler data structure analysis for improving locality in SMP systems and also for proving properties in software verification. In CARDS we extend SeaDSA [45] to detect disjoint data structures at compile time for far memory.

There is a also lot of work in prefetching in far memory such as Leap [92], Canvas [105], and 3PO [93]. These systems use several optimizations to prefetch complex access patterns and isolate application access patterns at runtime for far memory systems. However, unlike prior systems, CARDS uses static compiler analysis, which can simplify prefetching algorithms by guaranteeing isolation among data structures automatically at compile time. Similarly, compiler prefetching for recursive data structures has been explored for SMP systems [106, 107]. These compiler optimizations, however, rely on virtual memory support and do not directly apply to far memory.

Finally, the most related work in terms of far memory compilers is Mira [43], which requires profile traces to make the right policy decisions, and even to ensure correctness. CARDS is the first far memory compiler that shows how policy decisions can be made dynamically at data structure granularity for far memory architectures.

CHAPTER 7
CONTRIBUTIONS

Modeling speedup in software disaggregated environments We introduced speedup models to place bounds on application speedup along with a kernel module called mktrace to emulate communication overheads in multi-kernel systems. Our tools enable developers to gain insight whether an application benefits from resource disaggregation within a single server before using such a system, thereby improving developer productivity. This framework was introduced in [46] and we evaluate our tool, models on real OS disaggregated systems in [37]. More details on our speedup models, emulator can be found in Chapter 3. This work was published in IEEE MASCOTS © 2019 IEEE, and its extension was published in IEEE TPDS Volume 33, Issue 6, © 2021 IEEE.

TrackFM - Far out compiler support for a far memory world We introduce TrackFM a novel far memory compiler, which provides performance of state-of-the-art library based approaches with full compatibility like the kernel based approaches. This framework was introduced in [42]. TrackFM improves developer productivity by enabling developers to run unmodified code in memory disaggregated systems. More details on TrackFM can be found in Chapter 4. This work was published in ASPLOS 2024. This work is licensed under a Creative Commons Attribution International 4.0 License.

CARDS - Compiler assisted remote data structures for memory disaggregated servers Our design and implementation of CARDS will be the first far memory compiler to enable policy decisions dynamically with good performance. Our tool will open new areas of research in far memory such as where to place data structure memory objects based on memory pool device characteristics dynamically. More details on CARDS can be found in Chapter 5. We plan to submit this work to OSDI 2025.

CHAPTER 8

CONCLUSION AND FUTURE WORK

Software resource disaggregation has improved flexibility in data centers, however also requires developers to put additional effort to make their applications perform well. Moreover, there are no guarantees that after switching to a disaggregated system, the application will have good performance. My thesis explores automated techniques to improve developer productivity in disaggregated software stacks.

We first show that a developer can leverage our speedup models, emulator to gain insight whether an application benefits from resource disaggregation within single server [46]. After using our tool, the developer can decide based on our model whether such a system would be beneficial. We also evaluated our speedup models on real multi-OS systems [37].

We then build the TrackFM [42] compiler, which enables a developer to run applications efficiently on memory disaggregated systems without requiring developer changes. Our tool improves developer productivity in these systems by providing the transparency of kernel-based approaches and the performance of library-based approaches automatically using compiler analysis.

Finally, we build CARDS, which further eases developer productivity in memory disaggregated systems by making far memory policy decisions dynamically at data structure granularity. CARDS automatically transforms source-level data structures to far memory data structures to improve application performance by combining static and dynamic information. Furthermore, CARDS has the potential to simplify prefetching algorithms by isolating disjoint data structures automatically. Like our prior tools, we plan to make CARDS freely available online on upon publication. Our tool will enable researchers to explore new policy decisions at data structure granularity for far memory.

My thesis illustrates how low-level system software components, including compilers, operating systems, and analytical models, can significantly enhance developer productivity in software-based disaggregated systems. By building automated tools, we enable developers to concentrate on application logic, freeing them from concerns about resource management and optimization in disaggregated environments.

I now describe the key insights from my thesis. First, our modeling work in Chapter 3 demonstrates that even a simple model can yield valuable insights when a complex system is broken into performance-critical components, as long as parameters stay within certain thresholds. For instance, in Chapter 3, the naïve model achieves accuracy similar to the refined model when forwarding costs are limited to a few microseconds. A key finding from Chapter 4 is that eliminating instrumentation overhead, rather than merely reducing its cost, is a more effective strategy for amortization. Additionally, as network costs (e.g., object fetches from a remote server) increase, compiler-based approaches in disaggregated environments become more beneficial, with instrumentation overhead having less impact. However, when network costs decrease, instrumentation overhead begins to dominate. Chapter 5 shows that, in this scenario, selectively determining which data structures are remotable rather than marking all data structures as remotable allows application to achieve near-native performance.

Although my thesis offers foundational insights, these may not fully apply to managed languages. To address this, we plan to extend CARDS using a modular framework like MLIR [108], which could enable cross-compiler optimizations for languages like Java. Additionally, I plan to expand CARDS to emulate prototype hardware, providing an alternative to cycle-accurate simulators and offering valuable insights into application behavior at a higher level. For example, in future CXL-based systems [103], CARDS could optimize prefetching, caching, and data management for critical data structures, improving performance. Inspired by WARDen [76], CARDS could also reduce cache coherence traffic by identifying frequently accessed data structures. Moreover, integrating transformer-based machine learning models within CARDS could simplify automatic cross-platform application porting, boosting developer productivity in disaggregated environments.

BIBLIOGRAPHY

[1] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-performance, application-integrated far memory," in *Proceedings of the* 14^{th} *USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, (Berkeley, CA, USA), pp. 315–332, USENIX Association, Nov. 2020. https://www.usenix.org/conference/osdi20/presentation/ruan.

[2] Intel, "Intel rack scale design architecture," 2017. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

[3] HUAWEI, "High throughput computing data center architecture," 2014. http://acs.ict.ac.cn/asbd2014/202203/P020220329611085522455.pdf.

[4] M. Bielski, I. Syrigos, K. Katrinis, D. Syrivelis, A. Reale, D. Theodoropoulos, N. Alachiotis, D. Pnevmatikatos, E. Pap, G. Zervas, V. Mishra, A. Saljoghei, A. Rigo, J. F. Zazo, S. Lopez-Buedo, M. Torrents, F. Zyulkyarov, M. Enrico, and O. G. de Dios, "dReDBox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter," in *2018 Design, Automation and Test in Europe Conference and Exhibition*, pp. 1093–1098, 2018.

[5] K. Asanović, "FireBox: A hardware building block for 2020 Warehouse-Scale computers," Feb. 2014. USENIX FAST '14 Keynote Address.

[6] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), p. 267–278, Association for Computing Machinery, 2009. https://doi.org/10.1145/1555754.1555789.

[7] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?," in *Proceedings of the* 15^{th} *European Conference on Computer Systems*, EuroSys '20, (New York, NY, USA), Association for Computing Machinery, Apr. 2020. https://doi.org/10.1145/3342195.3387522.

[8] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with Infiniswap," in *Proceedings of the* 14^{th} *USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pp. 649–667, USENIX Association, Mar. 2017. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[9] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: A low overhead, software-only approach for supporting fine-grain shared memory," in *Proceedings of the* 7^{th} *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), pp. 174–185, Association for Computing Machinery, Oct. 1996. https://doi.org/10.1145/237090.237179.

[10] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: An architecture for extreme-scale operating systems," in *Proceedings of the* 4^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, pp. 2:1–2:8, June 2014.

[11] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa, "On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel," in *Proceedings of the* 30th *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16, pp. 1041–1050, May 2016.

[12] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "TMO: Transparent memory offloading in datacenters," in *Proceedings of the* 27th *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), p. 609–621, Association for Computing Machinery, Apr. 2022. https://doi.org/10.1145/3503222.3507731.

[13] W. Yoon, J. Ok, J. Oh, S. Moon, and Y. Kwon, "DiLOS: Do not trade compatibility for performance in memory disaggregation," in *Proceedings of the* 18th *European Conference on Computer Systems*, EuroSys '23, (New York, NY, USA), p. 266–282, Association for Computing Machinery, May 2023. https://doi.org/10.1145/3552326.3567488.

[14] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu, "Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, (Boston, MA), pp. 181–198, USENIX Association, Apr. 2023. https://www.usenix.org/conference/nsdi23/presentation/qiao.

[15] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: a disseminated, distributed OS for hardware resource disaggregation," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, (USA), p. 69–87, USENIX Association, 2018.

[16] K. C. Hale and P. A. Dinda, "A case for transforming parallel runtimes into operating system kernels," in *Proceedings of the* 24th *ACM Symposium on High-performance Parallel and Distributed Computing*, HPDC '15, June 2015.

[17] K. C. Hale and P. A. Dinda, "Enabling hybrid parallel runtimes through kernel and virtualization support," in *Proceedings of the* 12th *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pp. 161–175, Apr. 2016.

[18] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, Nov. 2008.

[19] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of OS noise," in *Proceedings of the* 25th *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pp. 852–863, May 2011.

[20] K. C. Hale and P. A. Dinda, "An evaluation of asynchronous events on modern hardware," in *Proceedings of the* 26th *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, Sept. 2018.

[21] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: Library operating systems for the cloud,” in *Proceedings of the* 18^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS’13, pp. 461–472, Mar. 2013.

[22] S. Lankes, S. Pickartz, and J. Breitbart, “HermitCore: A unikernel for extreme scale computing,” in *Proceedings of the* 6^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS’16, June 2016.

[23] S. Peter and T. Anderson, “Arrakis: A case for the end of the empire,” in *Proceedings of the* 14^{th} *Workshop on Hot Topics in Operating Systems*, HotOS XIII, May 2013.

[24] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti, “Achieving performance isolation with lightweight co-kernels,” in *Proceedings of the* 24^{th} *International Symposium on High-Performance Parallel and Distributed Computing*, HPDC ’15, pp. 149–160, June 2015.

[25] K. C. Hale, C. Hetland, and P. A. Dinda, “Multiverse: Easy conversion of runtime systems into os kernels via automatic hybridization,” in *Proceedings of the* 14^{th} *IEEE International Conference on Autonomic Computing*, ICAC’17, July 2017.

[26] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski, “Libra: A library operating system for a JVM in a virtualized execution environment,” in *Proceedings of the* 3^{rd} *International Conference on Virtual Execution Environments*, VEE ’07, pp. 44–54, June 2007.

[27] B. Kocoloski and J. Lange, “XEMEM: Efficient shared memory for composed applications on multi-OS/R exascale systems,” in *Proceedings of the* 24^{th} *International Symposium on High-Performance Parallel and Distributed Computing*, HPDC ’15, pp. 89–100, June 2015.

[28] A. Gara, M. A. Blumrich, D. Chen, G. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. M. Vranas, “Overview of the Blue Gene/L system architecture,” *IBM Journal of Research and Development*, vol. 49, pp. 195–212, Mar. 2005. https://doi.org/10.1147/rd.492.0195.

[29] B. Gerofi, R. Riesen, M. Takagi, T. Boku, K. Nakajima, Y. Ishikawa, and R. W. Wisniewski, “Performance and scalability of lightweight multi-kernel based operating systems,” in *Proceedings of the* 32^{nd} *IEEE International Parallel and Distributed Processing Symposium*, IPDPS ’18, pp. 116–125, May 2018.

[30] B. Gerofi, Y. Ishikawa, R. Riesen, R. W. Wisniewski, Y. Park, and B. Rosenburg, “A multi-kernel survey for high-performance computing,” in *Proceedings of the* 6^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS ’16, (New York, NY, USA), June 2016.

[31] R. Gioiosa, R. W. Wisniewski, R. Murty, and T. Inglett, “Analyzing system calls in multi-OS hierarchical environments,” in *Proceedings of the* 5^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS ’15, June 2015.

[32] Y. Park, E. V. Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. W. Wisniewski, "FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment," in *Proceedings of the IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pp. 211–218, Oct. 2012.

[33] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485, Apr. 1967.

[34] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, May 1988. https://doi.org/10.1145/42411.42415.

[35] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, SC '90, pp. 324–333, Nov. 1990. https://doi.org/10.1145/42411.42415.

[36] B. R. Tauro, C. Liu, and K. C. Hale, "Modeling speedup in multi-os environments," in *Proceedings of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '19, pp. 336–346, 2019. © 2019 IEEE.

[37] B. R. Tauro, C. Liu, and K. C. Hale, "Modeling speedup in multi-os environments," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2021. © 2021 IEEE.

[38] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote memory in the age of fast networks," in *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, (New York, NY, USA), pp. 121–127, Association for Computing Machinery, Sept. 2017. https://doi.org/10.1145/3127479.3131612.

[39] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *Proceedings of the 15th European Conference on Computer Systems*, EuroSys '20, (New York, NY, USA), Association for Computing Machinery, Apr. 2020. https://doi.org/10.1145/3342195.3387517.

[40] W. Yoon, J. Oh, J. Ok, S. Moon, and Y. Kwon, "DiLOS: Adding performance to paging-based memory disaggregation," in *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, (New York, NY, USA), pp. 70–78, Association for Computing Machinery, Aug. 2021. https://doi.org/10.1145/3476886.3477507.

[41] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI '19, (Berkeley, CA, USA), p. 361–377, USENIX Association, Feb. 2019.

[42] Tauro, Brian R. and Suchy, Brian and Campanoni, Simone and Dinda, Peter, and Hale, Kyle C., "TrackFM: Far-out compiler support for a far memory world," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '24, (New York, NY, USA), p. 401–419, Association for Computing Machinery, Apr. 2024. https://doi.org/10.1145/3617232.3624856.

[43] Z. Guo, Z. He, and Y. Zhang, "Mira: A program-behavior-guided far memory system," in *Proceedings of the* 29^{th} *Symposium on Operating Systems Principles*, SOSP '23, (New York, NY, USA), p. 692–708, Association for Computing Machinery, Oct. 2023. https://doi.org/10.1145/3600006.3613157.

[44] C. Lattner and V. Adve, "Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis," Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[45] A. Gurfinkel and J. A. Navas, "A context-sensitive memory model for verification of C/C++ programs," in *Static Analysis* (F. Ranzato, ed.), (Cham), pp. 148–168, Springer International Publishing, 2017.

[46] B. Tauro, C. Liu, and K. C. Hale, "Modeling speedup in multi-OS environments," in *Proceedings of the 27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '19, Oct. 2019.

[47] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the wily hacker," in *Proceedings of the* 6^{th} *USENIX Security Symposium*, SSYM '96, July 1996.

[48] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, pp. 151–180, Aug. 1998.

[49] K. Jain and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement," in *In Proceedings of the 2000 Network and Distributed System Security Symposium*, NDSS '00, Feb. 2000.

[50] N. Provos, "Improving host security with system call policies," in *Proceedings of the* 12^{th} *USENIX Security Symposium*, SSYM '03, Aug. 2003.

[51] K. C. Hale, L. Xia, and P. A. Dinda, "Shifting GEARS to enable guest-context virtual services," in *Proceedings of the* 9^{th} *International Conference on Autonomic Computing*, ICAC '12, pp. 23–32, Sept. 2012.

[52] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the* 15^{th} *ACM Conference on Computer and Communications Security*, CCS '08, pp. 51–62, Oct. 2008.

[53] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proceedings of the* 16^{th} *ACM Conference on Computer and Communications Security*, CCS '09, pp. 477–487, Nov. 2009.

[54] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, (New York, NY, USA), pp. 41–42, Association for Computing Machinery, 2018. https://doi.org/10.1145/3185768.3185771.

[55] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks-summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, SC '91, (New York, NY, USA), pp. 158–165, Association for Computing Machinery, Aug. 1991. https://doi.org/10.1145/125826.125925.

[56] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995. https://www.lammps.org/.

[57] K.-I. Ishikawa, Y. Kuramashi, A. Ukawa, and T. Boku, "CCS QCD miniapp," Mar. 2017. https://github.com/fiber-miniapp/ccs-qcd.

[58] A. Matni, E. A. Deiana, Y. Su, L. Gross, S. Ghosh, S. Apostolakis, Z. Xu, Z. Tan, I. Chaturvedi, B. Homerding, *et al.*, "NOELLE offers empowering LLVM extensions," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, pp. 179–192, IEEE, Apr. 2022.

[59] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, pp. 75–86, IEEE, Mar. 2004. https://doi.org/10.1109/CGO.2004.1281665.

[60] G. R. Manoj N. P., Manjunath K. V., "CAS-DSM: A compiler assisted software distributed shared memory," *International Journal of Parallel Programming*, vol. 32, no. 2, pp. 77–122, 2004.

[61] T. Matsumoto and K. Hiraki, "Memory-based communication facilities and asymmetric distributed shared memory," in *Proceedings of the* 1^{st} *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, IWIA '97, pp. 30–39, IEEE, Oct. 1997.

[62] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki, "Evaluation of compiler-assisted software DSM schemes for a workstation cluster," in *Proceedings of the* 3^{rd} *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, IWIA '99, pp. 57–68, IEEE, Nov. 1999.

[63] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '91, (New York, NY, USA), p. 30–44, Association for Computing Machinery, May 1991. https://doi.org/10.1145/113445.113449.

[64] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, (USA), p. 1–14, USENIX Association, July 2019. https://www.usenix.org/conference/atc19/presentation/duplyakin.

[65] J. D. McCalpin, "STREAM: Sustainable memory bandwidth in high performance computers," tech. rep., University of Virginia, 1991–2007.

[66] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004. https://www.linuxjournal.com/article/7451.

[67] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," Tech. Rep. NAS-95-020, NASA Ames Research Center, Dec. 1995. https://www.davidhbailey.com/dhbpapers/npb-2.0.pdf.

[68] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, (New York, NY, USA), p. 53–64, Association for Computing Machinery, 2012. https://doi.org/10.1145/2254756.2254766.

[69] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, "The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures," *Future Generation Computer Systems*, vol. 125, pp. 743–757, Dec. 2021.

[70] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), p. 222–233, Association for Computing Machinery, Oct. 1996. https://doi.org/10.1145/237090.237190.

[71] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, (New York, NY, USA), pp. 79–92, Association for Computing Machinery, Apr. 2021. https://doi.org/10.1145/3445814.3446713.

[72] *Userfaultfd*. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt.

[73] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), p. 66–78, Association for Computing Machinery, June 2015. https://doi.org/10.1145/2749469.2749471.

[74] Intel Corporation, *Intel® Architecture Instruction Set Extensions Programming Reference*, Dec. 2016.

[75] 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, (New York, NY, USA), p. 129–142, Association for Computing Machinery, June 2005. https://doi.org/10.1145/1065010.1065027.

[76] M. Wilkins, S. Westrick, V. Kandiah, A. Bernat, B. Suchy, E. A. Deiana, S. Campanoni, U. Acar, P. Dinda, and N. Hardavellas, "WARDen: Specializing cache coherence for high-level parallel languages," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '23, (New York, NY, USA), p. 122–135, Association for Computing Machinery, Mar. 2023. https://doi.org/10.1145/3579990.3580013.

[77] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of global address space languages: Co-array Fortran and Unified Parallel C," in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, (New York, NY, USA), p. 36–47, Association for Computing Machinery, 2005. https://dl.acm.org/doi/10.1145/1065944.1065950.

[78] M. Hofmann and S. Jost, "Static prediction of heap space usage for first-order functional programs," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, (New York, NY, USA), p. 185–197, Association for Computing Machinery, 2003. https://doi.org/10.1145/604131.604148.

[79] S. K. Muller and J. Hoffmann, "Modeling and analyzing evaluation cost of CUDA kernels," *Proceedings of the ACM on Programming Languages*, vol. 5, pp. 1–31, Jan. 2021. https://doi.org/10.1145/3434306.

[80] D.-J. Oh, Y. Moon, D. K. Ham, T. J. Ham, Y. Park, J. W. Lee, J. H. Ahn, and E. Lee, "MaPHeA: A framework for lightweight memory hierarchy-aware profile-guided heap allocation," *ACM Transactions on Embedded Computing Systems*, vol. 22, pp. 1–28, Dec. 2022. https://doi.org/10.1145/3527853.

[81] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal, "Designing far memory data structures: Think outside the box," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, (New York, NY, USA), p. 120–126, Association for Computing Machinery, 2019. https://doi.org/10.1145/3317550.3321433.

[82] 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, June 2005.

[83] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the 7^{th} International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), p. 222–233, Association for Computing Machinery, Oct. 1996. https://doi.org/10.1145/237090.237190.

[84] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *2012 Innovative Parallel Computing (InPar)*, pp. 1–10, 2012.

[85] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[86] L. Chen, S. Liu, C. Wang, H. Ma, Y. Qiao, Z. Wang, C. Wu, Y. Lu, X. Feng, H. Cui, S. Lu, and H. Xu, "A tale of two paths: Toward a hybrid data plane for efficient Far-Memory applications," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, (Santa Clara, CA), pp. 77–95, USENIX Association, July 2024. https://www.usenix.org/conference/osdi24/presentation/chen-lei.

[87] P. Kumar K., A. Sanyal, and A. Karkare, "Liveness-based garbage collection for lazy languages," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, (New York, NY, USA), p. 122–133, Association for Computing Machinery, 2016. https://doi.org/10.1145/2926697.2926698.

[88] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *Proceedings of the 4^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, May 1993.

[89] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, pp. 65–76, Apr. 2009.

[90] K. Z. Ibrahim, S. Williams, and L. Oliker, "Roofline scaling trajectories: A method for parallel application and architectural performance analysis," in *Proceedings of the International Conference on High Performance Computing and Simulation*, HPCS '18, pp. 350–358, July 2018.

[91] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proceedings of the* 27^{th} *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), p. 417–433, Association for Computing Machinery, Feb. 2022. https://doi.org/10.1145/3503222.3507762.

[92] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with Leap," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '20, (USA), pp. 843–857, USENIX Association, July 2020. https://www.usenix.org/conference/atc20/presentation/al-maruf.

[93] C. Branner-Augmon, N. Galstyan, S. Kumar, E. Amaro, A. Ousterhout, A. Panda, S. Ratnasamy, and S. Shenker, "3PO: Programmed far-memory prefetching for oblivious applications," 2022. https://arxiv.org/abs/2207.07688.

[94] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat, "Carbink: Fault-tolerant far memory," in *Proceedings of the* 16^{th} *USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, (Carlsbad, CA), pp. 55–71, USENIX Association, July 2022. https://www.usenix.org/conference/osdi22/presentation/zhou-yang.

[95] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv—optimizing the operating system for virtual machines," in *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC'14, June 2014.

[96] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," in *Proceedings of the* 6^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, (New York, NY, USA), p. 297–306, Association for Computing Machinery, Nov. 1994. https://doi.org/10.1145/195473.195575.

[97] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, "Software write detection for distributed shared memory," in *Proceedings of the* 1^{st} *Symposium on Operating Systems Design and Implementation*, OSDI '94, p. 8–es, USENIX Association, Nov. 1994. https://www.usenix.org/conference/osdi-94/software-write-detection-distributed-shared-memory.

[98] M. Cierniak and W. Li, "Unifying data and control transformations for distributed shared-memory machines," in *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '95, (New York, NY, USA), p. 205–217, Association for Computing Machinery, June 1995. https://doi.org/10.1145/207110.207145.

[99] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "An integrated compile-time/run-time software distributed shared memory system," in *Proceedings of the* 7^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), p. 186–197, Association for Computing Machinery, Sept. 1996. https://doi.org/10.1145/237090.237181.

[100] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel, "Compiler and software distributed shared memory support for irregular applications," in *Proceedings of the* 6^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '97, (New York, NY, USA), p. 48–56, Association for Computing Machinery, June 1997. https://doi.org/10.1145/263764.263772.

[101] B. Suchy, S. Campanoni, N. Hardavellas, and P. Dinda, "CARAT: A case for virtual memory through compiler- and runtime-based address translation," in *Proceedings of the* 41^{st} *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20, (New York, NY, USA), pp. 329–345, Association for Computing Machinery, June 2020. https://doi.org/10.1145/3385412.3385987.

[102] B. Suchy, S. Ghosh, D. Kersnar, S. Chai, Z. Huang, A. Nelson, M. Cuevas, A. Bernat, G. Chaudhary, N. Hardavellas, S. Campanoni, and P. Dinda, "CARAT CAKE: Replacing paging via compiler/kernel cooperation," in *Proceedings of the* 27^{th} *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), pp. 98–114, Association for Computing Machinery, Feb. 2022. https://doi.org/10.1145/3503222.3507771.

[103] "Compute express link (CXL)." `https://computeexpresslink.org/`. Accessed: 2024-1-11.

[104] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, (New York, NY, USA), p. 488–504, Association for Computing Machinery, 2021. https://doi.org/10.1145/3477132.3483561.

[105] C. Wang, Y. Qiao, H. Ma, S. Liu, W. Chen, R. Netravali, M. Kim, and G. H. Xu, "Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, (Boston, MA), pp. 161–179, USENIX Association, Apr. 2023. https://www.usenix.org/conference/nsdi23/presentation/wang-chenxi.

[106] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), p. 222–233, Association for Computing Machinery, 1996. https://doi.org/10.1145/237090.237190.

[107] C.-K. Luk and T. Mowry, "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 134–141, 1999.

[108] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, p. 2–14, IEEE Press, 2021. https://doi.org/10.1109/CGO51591.2021.9370308.